SAND2001-3796
Unlimited Release
April 2002
Updated April 2003
Updated Fall 2005

DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis

Version 4.0 User's Manual

Michael S. Eldred, Shannon L. Brown, Laura P. Swiler, Brian M. Adams, Daniel M. Dunlavy, David M. Gay
Optimization and Uncertainty Estimation Department

Anthony A. Giunta
Validation and Uncertainty Quantification Processes Department

William E. Hart, Jean-Paul Watson
Discrete Algorithms and Math Department

John P. Eddy
System Sustainment and Readiness Technologies Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0370

Josh D. Griffin, Patty D. Hough, Tamara G. Kolda, Monica L. Martinez-Canales, Pamela J. Williams
Computational Sciences and Mathematics Research Department

Sandia National Laboratories
P.O. Box 969
Livermore, California 94551-0969

Abstract

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible and extensible interface between simulation codes and iterative analysis methods. DAKOTA con-

tains algorithms for optimization with gradient and nongradient-based methods; uncertainty quantification with sampling, reliability, and stochastic finite element methods; parameter estimation with nonlinear least squares methods; and sensitivity analysis with design of experiments and parameter study methods. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for design and performance analysis of computational models on high performance computers.

This report serves as a user's manual for the DAKOTA software and provides capability overviews and procedures for software execution, as well as a variety of example studies.

Contents

Preface

Preface to DAKOTA Version 3.3 Users Manual

For version 3.3 of DAKOTA, we have added significant enhancements in the following areas:

- major redesign of multilevel parallelism classes to allow multiple parallel configurations to coexist. Previously, only a sequence of partitions could be used, one at a time. This allows more complete parallelism management in strategies which have more than one active model at a time, such as multifidelity SBO.
- variance-based decomposition (VBD) sensitivity analysis method. VBD provides global sensitivity indices for each uncertain variable.
- capability for 2nd-order probability where one can specify an outer, epistemic level of uncertainty, and inner, aleatory level of uncertainty. Sample realizations of uncertain variables at the outer level are then inserted into the distribution parameters for an uncertainty analysis at the aleatory level.
- addition of finite-difference numerical Hessians and BFGS/SR1 quasi-Newton Hessians to DAKOTA's derivative estimation routines.
- new Acro release. Acro is a global optimization library. Dakota now includes updated COLINY methods supporting general nonlinear constraints: APPS, DIRECT, Solis-Wets, Pattern search, and Cobyla.
- new quasi-Monte Carlo sampling methods, accessible through the FSUDace library. The quasi-Monte Carlo methods currently available are Hammersley and Halton sequences. Centroidal Voronoi Tesselation (CVT) is also available. These methods are constructed to provide good uniformity coverage of samples throughout the unit hypercube for design of computer experiments.

Preface to DAKOTA Version 3.2 Users Manual

For version 3.2 of DAKOTA, we have added significant enhancements in the following areas:

- multiobjective optimization without weights via genetic algorithms,
- a revised penalty function formulation for surrogate based optimization,
- full- and quasi-Newton second order correction methods for surrogate based optimization,
- a secant-based nonlinear least squares solution algorithm via NL2SOL,
- a simplified interface to the OPT++ optimization methods, along with the addition of nonlinear interior point methods to the OPT++ library of solvers,
- nondeterministic sampling results output in cumulative or complementary cumulative formats, plus output of simple correlations, partial correlations, and rank correlations,

- nondeterministic reliability methods that allow user-specified probability levels or reliability levels, and either a first- or second-order integration scheme, and
- the DPREPRO utility to automate the “cut-and-paste” steps of transferring the parameter values output by DAKOTA into the input file for a user’s simulation code.

Future versions of DAKOTA will incorporate enhancements to the nondeterministic methods area, such as quasi-Monte Carlo sampling and related approaches, a new interface to the COLINY package of evolutionary and non-gradient optimization algorithms, and a new graphical user interface to streamline the creation of DAKOTA input files and the interpretation of DAKOTA output files.

Preface to DAKOTA Version 3.1 Users Manual

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) project started in 1994 as an internal research and development activity at Sandia National Laboratories in Albuquerque, New Mexico. The original goal of this effort was to provide a common set of optimization tools for a group of engineers who were solving structural analysis and design problems. Prior to the start of the DAKOTA project, there was not a focused effort to archive the optimization methods for reuse on other projects. Thus, for each new project the engineers found themselves custom building new interfaces between the engineering analysis software and the optimization software. This was a particular burden when attempts were made to use parallel computing resources, where each project required the development of a unique master program that coordinated concurrent simulations on a network of workstations or a parallel computer. The initial DAKOTA toolkit provided the engineering and analysis community at Sandia Labs with access to a variety of different optimization methods and algorithms, with much of the complexity of the optimization software interfaces hidden from the user. Thus, the engineers were easily able to switch between optimization software packages simply by changing a few lines in the DAKOTA input file. In addition to applications in structural analysis, DAKOTA has been applied to applications in computational fluid dynamics, nonlinear dynamics, shock physics, heat transfer, and many others.

DAKOTA has grown significantly beyond its original focus as a toolkit of optimization methods. In addition to having many state-of-the-art optimization methods, DAKOTA now includes methods for sensitivity analysis, parameter estimation, design-of-experiments, uncertainty quantification, and multidimensional surface mapping. Underlying all of these methods is support for parallel computation; ranging from the level of a desktop multiprocessor computer up to massively parallel computers found at national laboratories and supercomputer centers.

The objective of the public release of the DAKOTA software is to facilitate collaborations among the developers of DAKOTA at Sandia National Laboratories and other institutions, including academic, governmental, and corporate entities. We are interested in developing relationships with persons or groups who would like to assist us in extending the capabilities of DAKOTA. We feel that this goal is best pursued by making the source code of our software freely available to others. In doing so, we expect that some of our errors will be found and corrected, and that new capabilities will be added to future versions of DAKOTA. Currently, DAKOTA is licensed for public release under a GNU General Public License. See http://www.gnu.org/licenses/gpl.html for more information on the GPL software use agreement.

The core DAKOTA framework developers are Mike Eldred, Tony Giunta, Laura Swiler, David Gay, Steve Wojtkiewicz, and Shane Brown. In addition, Bill Hart, Jean-Paul Watson, and Pam Williams develop and maintain DAKOTA’s interfaces to the COLINY, PICO, UTILIB, OPT++, DDACE, and APPS libraries and John Eddy develops and maintains the JEGA library. Additional contributors to these libraries include Patty Hough, Tammy Kolda, Monica Martinez-Canales, Cindy Phillips, and John Red-Horse from Sandia; as well as Prof. Roger Ghanem from Johns Hopkins University; Prof. Jonathan Eckstein from Rutgers University; and Prof. Virginia Torczon from the College of William and Mary.

Contact Information:

Michael Eldred, Principal Investigator - DAKOTA Project
Sandia National Laboratories

P.O. Box 5800
Mail Stop 0370
Albuquerque, NM 87185-0370

email: <dakota@sandia.gov>
web: http://endo.sandia.gov/DAKOTA

Chapter 1

Introduction

1.1 Motivation for DAKOTA Development

Computational models are commonly used in engineering design activities for simulating complex physical systems in disciplines such as fluid mechanics, structural dynamics, heat transfer, nonlinear structural mechanics, shock physics, and many others. These simulators can be an enormous aid to engineers who want to develop an understanding and/or predictive capability for the complex behaviors that are often observed in the respective physical systems. Often, these simulators are employed as virtual prototypes, where a set of predefined system parameters, such as size or location dimensions and material properties, are adjusted to improve or optimize the performance of a particular system, as defined by one or more system performance objectives. Optimization of the virtual prototype then requires execution of the simulator, evaluation of the performance objective(s), and adjustment of the system parameters in an iterative and directed way, such that an improved or optimal solution is obtained for the simulation as measured by the performance objective(s). System performance objectives can be formulated, for example, to minimize weight, cost, or defects; to limit a critical temperature, stress, or vibration response; or to maximize performance, reliability, throughput, agility, or design robustness.

One of the primary motivations for the development of DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) has been to provide engineers with a systematic and rapid means of obtaining improved or optimal designs using their simulator-based models. Making this capability available to engineers generally leads to better designs and improved system performance at earlier stages of the design phase, and eliminates some of the dependence on real prototypes and testing, thereby shortening the design cycle and reducing overall product development costs. In addition to providing this environment for answering systems performance questions, the DAKOTA toolkit also provides an extensible platform for the research and rapid prototyping of customized methods and strategies [23].

1.2 Capabilities of DAKOTA

The DAKOTA toolkit provides a flexible, extensible interface between your simulation code and a variety of iterative methods and strategies. While DAKOTA was originally conceived as an easy-to-use interface between simulation codes and optimization algorithms, recent versions have been expanded to interface with other types of iterative analysis methods such as uncertainty quantification with nondeterministic propagation methods, parameter estimation with nonlinear least squares solution methods, and sensitivity analysis with general-purpose design of experiments and parameter study capabilities. These capabilities may be used on their own or as building blocks within more sophisticated strategies such as hybrid optimization, surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty.

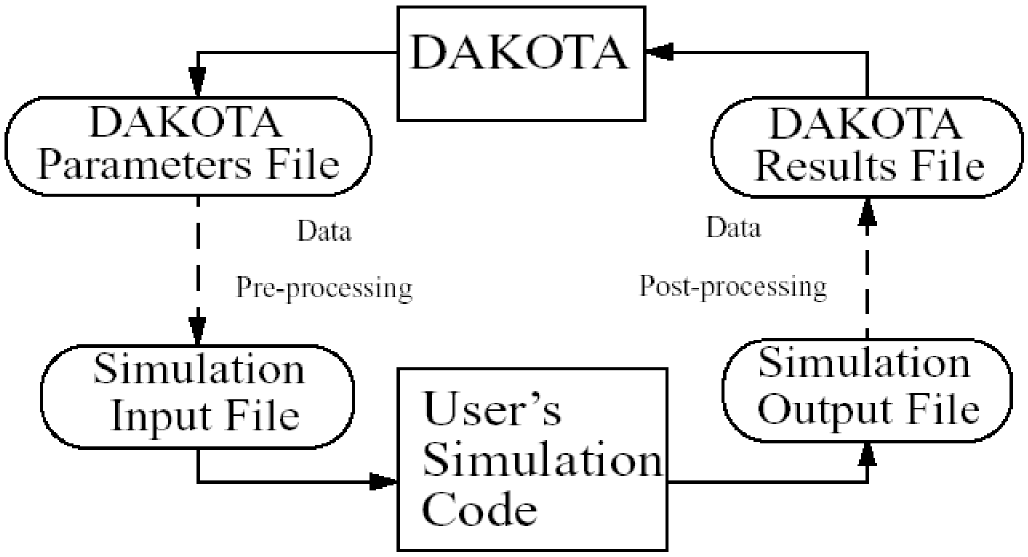


Figure 1.1: The loosely-coupled or “black-box” interface between DAKOTA and a user-supplied simulation code.

Thus, one of the primary advantages that DAKOTA has to offer is that access to a very broad range of iterative capabilities can be obtained through a single, relatively simple interface between DAKOTA and your simulator. Should you want to try a different type of iterative method or strategy with your simulator, it is only necessary to change a few commands in the DAKOTA input and start a new analysis. The need to learn a completely different style of command syntax and the need to construct a new interface each time you want to use a new algorithm are eliminated.

1.3 How Does DAKOTA Work?

Figure 1.1 depicts the loosely-coupled, or “black-box,” relationship between DAKOTA and the simulation code(s). This loose coupling is the simplest approach and is the one that most DAKOTA users will employ. Data is exchanged between DAKOTA and the simulation code by reading and writing short data files, and DAKOTA does not require access to the source code of the user’s simulation software. DAKOTA is executed using commands that the user supplies in an input file (not shown in Figure 1.1) which specify the type of analysis to be performed (e.g., parameter study, optimization, uncertainty estimation, etc.), along with the file names associated with the user’s simulation code. During its operation, DAKOTA automatically executes the user’s simulation code by creating a separate UNIX process that is external to DAKOTA.

The solid lines in Figure 1.1 denote file input/output (I/O) operations that are part of DAKOTA or the user’s simulation code. The dotted lines indicate the passing of information that must be handled by the user. As DAKOTA is running, it writes out a parameters file that contains the values of the current variables. DAKOTA then starts the user’s simulation code (or, often, a short driver script), and when the simulation has completed, DAKOTA reads in the response data from a results file. This process is repeated until all of the simulation code runs required by the iterative study have been completed.

In some cases it is advantageous to have a close coupling between DAKOTA and the user’s simulation code. This close coupling is an advanced feature of DAKOTA and is accomplished through either a direct interface or a SAND (simultaneous analysis and design) interface. For the direct interface, the user’s simulation code is modified to behave as a function or subroutine under DAKOTA. This interface can be considered to be “semi-intrusive” in that it requires relatively minor modifications to the simulation code.

Its major advantage is the elimination of the overhead resulting from file I/O and UNIX process creation. It can also be a useful tool for parallel processing, by encapsulating everything within a single executable. The SAND interface approach is “fully intrusive” in that it requires further modifications to the simulation code so that DAKOTA has access to the internal vectors and matrices computed by the simulation code. With the SAND approach, both the optimization method in DAKOTA and a nonlinear simulation code are converged simultaneously. While this approach can greatly reduce the computational expense of optimization, considerable software development effort must be expended to achieve this intrusive coupling between SAND optimization methods and the simulation code.

1.4 Background and Mathematical Formulations

This section provides a basic introduction to the mathematical formulation of optimization, nonlinear least squares, sensitivity analysis, design of experiments, and uncertainty quantification problems. The primary goal of this section is to introduce terms relating to these topics, and is not intended to be a description of theory or numerical algorithms. There are numerous sources of information on these topics ([3], [32], [40], [41], [55], [66]) and the interested reader is advised to consult one or more of these texts.

1.4.1 Optimization

A general optimization problem is formulated as follows:

$$\begin{aligned}
\text{minimize:} \quad & f(\mathbf{x}) \\
& \mathbf{x} \in \Re^n \\
\text{subject to:} \quad & \mathbf{g}_L \leq \mathbf{g}(\mathbf{x}) \leq \mathbf{g}_U \\
& \mathbf{h}(\mathbf{x}) = \mathbf{h}_t \\
& \mathbf{a}_L \leq \mathbf{A}_i\mathbf{x} \leq \mathbf{a}_U \\
& \mathbf{A}_e\mathbf{x} = \mathbf{a}_t \\
& \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U
\end{aligned} \tag{1.1}$$

where vector and matrix terms are marked in bold typeface. In this formulation, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is an n-dimensional vector of real-valued *design variables* or *design parameters*. The n-dimensional vectors, $\mathbf{x}_L$ and $\mathbf{x}_U$, are the lower and upper bounds, respectively, on the design parameters. These bounds define the allowable values for the elements of $\mathbf{x}$, and the set of all allowable values is termed the *design space* or the *parameter space*. A *design point* or a *sample point* is a set of values for that fall within the parameter space.

The optimization goal is to minimize the *objective function*, $f(\mathbf{x})$, while satisfying the constraints. Constraints can be categorized as either linear or nonlinear and as either inequality or equality. The *nonlinear inequality constraints*, $\mathbf{g}(\mathbf{x})$, are “2-sided,” in that they have both lower and upper bounds $\mathbf{g}_L$, and $\mathbf{g}_U$, respectively. The *nonlinear equality constraints*, $\mathbf{h}(\mathbf{x})$, have target values specified by $\mathbf{h}_t$. The linear inequality constraints create a linear system $\mathbf{A}_i\mathbf{x}$, where $\mathbf{A}_i$ is the coefficient matrix for the linear system. These constraints are also 2-sided as they have and as lower and upper bounds, respectively. The linear equality constraints create a linear system $\mathbf{A}_e\mathbf{x}$, where $\mathbf{A}_e$ is the coefficient matrix for the linear system and are the target values. The constraints partition the parameter space into feasible and infeasible regions. A design point is said to be *feasible* if and only if it satisfies all of the constraints. Correspondingly, a design point is said to be *infeasible* if it violates one or more of the constraints.

Many different methods exist to solve the optimization problem given by Equation 1.1, all of which iterate on $\mathbf{x}$ in some manner. That is, an initial value for each parameter in $\mathbf{x}$ is chosen, the *response* *quantities*, $f(\mathbf{x})$, $\mathbf{g}(\mathbf{x})$, $\mathbf{h}(\mathbf{x})$, are computed, and some algorithm is applied to generate a new $\mathbf{x}$ that will either reduce the objective function, reduce the amount of infeasibility, or both. To facilitate a general presentation of these methods, three criteria will be used in the following discussion to differentiate them: optimization problem type, search goal, and search method.

The optimization problem type can be characterized both by the types of constraints present in the problem and by the linearity or nonlinearity of the objective and constraint functions. For constraint categorization, a hierarchy of complexity exists for optimization algorithms, ranging from simple bound constraints, through linear constraints, to full nonlinear constraints. By the nature of this increasing complexity, optimization problem categorizations are inclusive of all constraint types up to a particular level of complexity. That is, an *unconstrained problem* has no constraints, a *bound-constrained problem* has only lower and upper bounds on the design parameters, a *linearly-constrained problem* has both linear and bound constraints, and a *nonlinearly-constrained problem* may contain the full range of nonlinear, linear, and bound constraints. If all of the linear and nonlinear constraints are equality constraints, then this is referred to as an *equality-constrained problem*, and if all of the linear and nonlinear constraints are inequality constraints, then this is referred to as an *inequality-constrained problem*. Further categorizations can be made based on the linearity of the objective and constraint functions. A problem where the objective function and all constraints are linear is called a *linear programming (LP) problem*. These types of problems commonly arise in scheduling, logistics, and resource allocation applications. Likewise, a problem where at least some of the objective and constraint functions are nonlinear is called a *nonlinear programming (NLP) problem*. These NLP problems predominate in engineering applications and are the primary focus of DAKOTA.

The search goal refers to the ultimate objective of the optimization algorithm, i.e., either global or local optimization. In *global optimization*, the goal is to find the design point that gives the lowest feasible objective function value over the entire parameter space. In contrast, in *local optimization*, the goal is to find a design point that is lowest relative to a “nearby” region of the parameter space. In almost all cases, global optimization will be more computationally expensive than local optimization. Thus, the user must choose an optimization algorithm with an appropriate search scope that best fits the problem goals and the computational budget.

The search method refers to the approach taken in the optimization algorithm to locate a new design point that has a lower objective function or is more feasible than the current design point. The search method can be classified as either *gradient-based* or *nongradient-based*. In a gradient-based algorithm, gradients of the response functions are computed to find the direction of improvement. Gradient-based optimization is the search method that underlies many efficient local optimization methods. However, a drawback to this approach is that gradients can be computationally expensive, inaccurate, or even nonexistent. In such situations, nongradient-based search methods may be useful. There are numerous approaches to nongradient-based optimization. Some of the more well known of these include pattern search methods (nongradient-based local techniques) and genetic algorithms (nongradient-based global techniques).

The overview of optimization methods presented above underscores that there is no single optimization method or algorithm that works best for all types of optimization problems. Chapter 17 provides some guidelines on choosing which DAKOTA optimization algorithm is best matched to your specific optimization problem.

1.4.2 Nonlinear Least Squares for Parameter Estimation

Specialized least squares solution algorithms can exploit the structure of a sum of the squares objective function for problems of the form:

$$\begin{aligned} \text{minimize:} \quad & f(\mathbf{x}) = \sum_{i=1}^{n} [T_i(\mathbf{x})]^2 \\ & \mathbf{x} \in \Re^n \end{aligned}$$

$$\begin{aligned} \text{subject to:} \quad & \mathbf{g}_L \leq \mathbf{g}(\mathbf{x}) \leq \mathbf{g}_U \\ & \mathbf{h}(\mathbf{x}) = \mathbf{h}_t \\ & \mathbf{a}_L \leq \mathbf{A}_i \mathbf{x} \leq \mathbf{a}_U \\ & \mathbf{A}_e \mathbf{x} = \mathbf{a}_t \\ & \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U \end{aligned} \tag{1.2}$$

where $f(\mathbf{x})$ is the objective function to be minimized and $T_i(\mathbf{x})$ is the ith least squares term. The bound, linear, and nonlinear constraints are the same as described previously for (1.1). Specialized least squares algorithms are generally based on the Gauss-Newton approximation. When differentiating $f(\mathbf{x})$ twice, terms of $T_i(\mathbf{x})T_i''(\mathbf{x})$ and $[T_i'(\mathbf{x})]^2$ result. By assuming that the former term tends toward zero near the solution since $T_i(\mathbf{x})$ tends toward zero, then the Hessian matrix of second derivatives of $f(\mathbf{x})$ can be approximated using only first derivatives of $T_i(\mathbf{x})$. As a result, Gauss-Newton algorithms exhibit quadratic convergence rates near the solution for those cases when the Hessian approximation is accurate, i.e. the residuals tend towards zero at the solution. Thus, by exploiting the structure of the problem, the second order convergence characteristics of a full Newton algorithm can be obtained using only first order information from the least squares terms.

A common example for $T_i(\mathbf{x})$ might be the difference between experimental data and model predictions for a response quantity at a particular location and/or time step, i.e.:

$$T_i(\mathbf{x}) = R_i(\mathbf{x}) - \overline{R_i} \tag{1.3}$$

where $R_i(\mathbf{x})$ is the response quantity predicted by the model and $\overline{R_i}$ is the corresponding experimental data. In this case, $\mathbf{x}$ would have the meaning of model parameters which are not precisely known and are being calibrated to match available data. This class of problem is known by the terms parameter estimation, system identification, model calibration, test/analysis reconciliation, etc.

1.4.3 Sensitivity Analysis and Parameter Studies

In many engineering design applications, sensitivity analysis techniques and parameter study methods are useful in identifying which of the design parameters have the most influence on the response quantities. This information is helpful prior to an optimization study as it can be used to remove design parameters that do not strongly influence the responses. In addition, these techniques can provide assessments as to the behavior of the response functions (smooth or nonsmooth, unimodal or multimodal) which can be invaluable in algorithm selection for optimization, uncertainty quantification, and related methods. In a post-optimization role, sensitivity information is useful is determining whether or not the response functions are robust with respect to small changes in the optimum design point.

In some instances, the term sensitivity analysis is used in a local sense to denote the computation of response derivatives at a point. These derivatives are then used in a simple analysis to make design decisions. DAKOTA supports this type of study through numerical finite-differencing or retrieval of analytic gradients computed within the analysis code. The desired gradient data is specified in the responses section of the DAKOTA input file and the collection of this data at a single point is accomplished through a parameter study method with no steps. This approach to sensitivity analysis should be distinguished from the activity of augmenting analysis codes to internally compute derivatives using techniques such as direct or adjoint differentiation, automatic differentiation (e.g., ADIFOR), or complex step modifications. These sensitivity augmentation activities are completely separate from DAKOTA and are outside the scope of this manual. However, once completed, DAKOTA can utilize these analytic gradients to perform optimization, uncertainty quantification, and related studies more reliably and efficiently.

In other instances, the term sensitivity analysis is used in a more global sense to denote the investigation of variability in the response functions. DAKOTA supports this type of study through computation of response data sets (typically function values only, but all data sets are supported) at a series of points in the parameter space. The series of points is defined using either a vector, list, centered, or multidimensional parameter study method. For example, a set of closely-spaced points in a vector parameter study could be used to assess the smoothness of the response functions in order to select a finite difference step size, and a set of more widely-spaced points in a centered or multidimensional parameter study could be used to determine whether the response function variation is likely to be unimodal or multimodal. See Chapter 8 for additional information on these methods. These more global approaches to sensitivity analysis can be used to obtain trend data even in situations when gradients are unavailable or unreliable, and they are conceptually similar to the design of experiments methods and sampling approaches to uncertainty quantification described in the following sections.

1.4.4 Design of Experiments

Classical design of experiments (DoE) methods and the more modern design and analysis of computer experiments (DACE) methods are both techniques which seek to extract as much trend data from a parameter space as possible using a limited number of sample points. Classical DoE techniques arose from technical disciplines that assumed some randomness and nonrepeatability in field experiments (e.g., agricultural yield, experimental chemistry). DoE approaches such as central composite design, Box-Behnken design, and full and fractional factorial design generally put sample points at the extremes of the parameter space, since these designs offer more reliable trend extraction in the presence of nonrepeatability. DACE methods are distinguished from DoE methods in that the nonrepeatability component can be omitted since computer simulations are involved. In these cases, space filling designs such as orthogonal array sampling and latin hypercube sampling are more commonly employed in order to accurately extract trend information. Quasi-Monte Carlo sampling techniques which are constructed to fill the unit hypercube with good uniformity of coverage can also be used for DACE.

DAKOTA supports both DoE and DACE techniques. In common usage, only parameter bounds are used in selecting the samples within the parameter space. Thus, DoE and DACE can be viewed as special cases of the more general probabilistic sampling for uncertainty quantification (see following section), in which the DoE/DACE parameters are treated as having uniform probability distributions. The DoE/DACE techniques are commonly used for investigation of global response trends, identification of significant parameters (e.g., main effects), and as data generation methods for building response surface approximations.

1.4.5 Uncertainty Quantification

Uncertainty quantification (UQ) is related to sensitivity analysis in that the common goal is to gain an understanding of how variations in the parameters affect the response functions of the engineering design problem. However, for uncertainty quantification, some or all of the components of the parameter vector, $\mathbf{x}$, are considered to be uncertain and not precisely known. The uncertain parameter values are specified by a probability distribution (e.g., normal/Gaussian) rather than a unique value.

The impact on the response functions due to the probabilistic nature of the parameters is often estimated using a sampling-based approach such as Monte Carlo sampling or one of its variants (latin hypercube, quasi-Monte Carlo, Markov-chain Monte Carlo, etc.). In these sampling approaches, a random number generator is used to select different values of the parameters with probability specified by their probability distributions. This is the point that distinguishes UQ sampling from DoE/DACE sampling, in that the former supports general probabilistic descriptions of the parameter set and the latter generally supports only a bounded parameter space description (i.e., uniform probabilities). A particular set of parameter values is often called a *sample point*, or simply a *sample*. After a user-selected number of sample points has been generated, the response functions for each sample are evaluated. Then, a statistical analysis is performed on the response function values to yield information on their characteristics. While this approach is straightforward, and readily amenable to parallel computing, it can be computationally expensive depending on the accuracy requirements of the statistical information (which links directly to the number of sample points).

When sampling methods are too expensive to apply, various analytic and quasi-analytic reliability methods can be applied to UQ problems. These include the Advanced Mean Value (AMV) and AMV+ algorithms, along with the first-order reliability method (FORM) and the second-order reliability method (SORM) [41]. These techniques all solve internal optimization problems in order to locate the most probable point (MPP) of failure. The MPP is then used as the point about which approximate probabilities are integrated.

In addition, stochastic finite element (SFE) approaches using polynomial chaos expansions are also available for characterizing the response of systems whose governing equations involve stochastic coefficients. The sampling, analytic reliability, and SFE approaches are described in more detail in Chapter 10.

1.5 Using this Manual

The previous sections in this chapter have provided a brief overview of the capabilities in DAKOTA, and have introduced some of the common terms that are used in the fields of optimization, parameter estimation, sensitivity analysis, design of experiments, and uncertainty quantification. The DAKOTA user that is new to these techniques is advised to consult the references cited earlier in this chapter to obtain more detailed descriptions of methods and algorithms in these disciplines.

Chapter 2 provides information on how to obtain, install, and use DAKOTA. In addition, example problems are presented in this chapter to demonstrate some of DAKOTA's capabilities for parameter studies, optimization, and UQ. Chapter 3 provides a brief overview of all of the different software packages and capabilities in DAKOTA. Chapter 4 through Chapter 6 provide information on model components which are involved in parameter to response mappings and Chapter 7 describes the output created by DAKOTA. Chapter 8 through Chapter 12 provide details on the iterative algorithms supported in DAKOTA, and Chapter 13 describes DAKOTA's advanced optimization strategies. Chapter 14 describes the approximation methods available in DAKOTA, Chapter 15 covers DAKOTA's parallel computing capabilities, Chapter 16 provides information on interfacing DAKOTA with engineering simulation codes, and Chapter 17 provides some usage guidelines for selecting DAKOTA algorithms. Finally, Chapter 18 through Chapter 20 describe restart utilities, failure capturing facilities, and additional test problems, respectively.

Chapter 2

Getting Started with DAKOTA

2.1 Installation Guide

DAKOTA can be compiled for most common computer systems that run the UNIX and LINUX operating systems. The computers and operating systems actively supported by the DAKOTA project include:

- Sun Solaris 2.8
- SGI IRIX 6.5
- Compaq/DEC OSF 5.1
- IBM AIX 5.2
- Intel PC Redhat LINUX 9
- ASCI Red

In addition, partial support is provided for Cplant, PC Windows (via Cygwin), Mac OSX, and HP HPUX. Additional details are provided in the file `/Dakota/README` in the distribution (see the following section for download instructions).

2.1.1 How to Obtain DAKOTA - External to Sandia Labs

If you are outside of Sandia National Laboratories, the DAKOTA binary executable files and source code files are available through the following web site:

http://endo.sandia.gov/DAKOTA

To receive the binary or source code files, you are asked to fill out a short online registration form. This information will be used by the DAKOTA development team to collect software usage metrics and, if desired, to register you for update announcements.

If you are a new DAKOTA user, we suggest that you download one of the binary executable distributions rather than the source code distribution. The compilation process can be somewhat involved, and it will be easier for you to first gain an understanding of DAKOTA by running the example problems that are provided with one of DAKOTA's binary distributions. For more experienced users, DAKOTA can be customized with additional packages and ported to additional computer platforms when building from the source code.

2.1.2 How to Obtain DAKOTA - Internal to Sandia Labs

DAKOTA binary executable files have been compiled and distributed to SCICO LAN and common compute servers at Sandia, Los Alamos, and Lawrence Livermore. Common locations include `/usr/local/bin/dakota` and `/projects/dakota/bin/<system>/dakota`, where "`<system>`" is `osf`, `irix`, `tflop` (ASCI Red service node), or `cougar` (ASCI Red compute node). Note that on some systems (e.g., ASCI Red), `/Net` may precede `/projects` due to NFS mounting of file systems. To see if DAKOTA is available on your computer system and accessible in your UNIX environment path settings, type the command `which dakota` at the UNIX prompt. If the DAKOTA executable file is in your path, its location will be echoed to the terminal. If the DAKOTA executable file is available on your system but not in your path, then you will need to locate it and add its directory to your path (the UNIX `whereis` and `find` commands can be useful for locating the executable).

If DAKOTA is not available on your system, the current preferred options are to either get an account on one of the common compute servers where DAKOTA is maintained, or if this is not practical, contact one of the DAKOTA team members so that we can provide you with DAKOTA executable files that are as complete as possible (i.e., that include Sandia-specific and site-licensed software that is not yet publicly available). Alternatively, you can follow the instructions given in the previous section to obtain the public version of the DAKOTA binary and/or source codes files. In the future, a download facility on Sandia's internal restricted network may be added to simplify internal distributions.

2.1.3 Installing DAKOTA - Binary Executable Files

Once you have downloaded a binary distribution from the web site listed above, you will have a UNIX tar file that has a name similar to `Dakota_3_x.OSversion.tar.gz`.

[Note to Windows Users: Some users have found that the name of the tar file gets corrupted when downloading the tar file to a PC running Windows. Before proceeding, verify that the name of the downloaded tar file is the same as the name listed on the DAKOTA web site. If the file name has been corrupted, rename it before attempting the steps listed below.]

Use the UNIX utility `gunzip` to uncompress the tar file and the UNIX `tar` utility to extract the files from the archive by executing the following commands:

```
gunzip Dakota_3_x.OSversion.tar.gz
tar -xvf Dakota_3_x.OSversion.tar
```

The tar utility will create a subdirectory named `/Dakota` in which the DAKOTA executables and example files will be stored. The executables are in `/Dakota/bin`, and the example problems are in `/Dakota/GettingStarted/Examples` and in `/Dakota/test`.

2.1.4 Installing DAKOTA - Source Code Files

The installation process for the DAKOTA source code files is more involved than the installation process for the binary files. Detailed instructions for installing DAKOTA are given in the file `/Dakota/INSTALL`.

2.1.5 Running DAKOTA

The DAKOTA executable file is named dakota. If this command is entered at the UNIX prompt without any arguments, the following usage message is returned to the user:

```
usage: dakota [options and <args>]
        -help (Print this summary)
```

```
            -version (Print DAKOTA version number)
            -input <$val> (REQUIRED DAKOTA input file $val)
            -output <$val> (Redirect DAKOTA standard output to file $val)
            -error <$val> (Redirect DAKOTA standard error to file $val)
            -read_restart <$val> (Read an existing DAKOTA restart file $val)
            -stop_restart <$val> (Stop restart file processing at evaluation $val)
            -write_restart <$val> (Write a new DAKOTA restart file $val)
```

Of these available command line inputs, only the "`-input`" option is required; all others are optional. The "`-help`" option prints the usage message above. The "`-version`" option prints the version number of the executable. The "`-input`" option provides the name of the DAKOTA input file. The "`-output`" and "`-error`" options provide file names for redirection of the DAKOTA standard output (stdout) and standard error (stderr), respectively. The "`-read_restart`" and "`-write_restart`" command line inputs provide the names of restart databases to read from and write to, respectively. The "`-stop_restart`" command line input limits the number of function evaluations read from the restart database (the default is all the evaluations) for those cases in which some evaluations were erroneous or corrupted. Restart management is an important technique for retaining data from expensive engineering applications. This is an advanced topic that is discussed in detail in Chapter 17. Note that these command line inputs can be abbreviated so long as the abbreviation is unique (the current set of command line options do not have any possibility for abbreviation ambiguity). That is, "`-h`", "`-v`", "`-i`", "`-o`", "`-e`", "`-r`", "`-s`", and "`-w`" are commonly used in place of the longer forms of the command line inputs.

To run DAKOTA with a particular input file, the following syntax can be used:

```
    dakota -i dakota.in
```

This will echo the standard output (stdout) and standard error (stderr) messages to the terminal. To redirect stdout and stderr to separate files, the `-o` and `-e` command line options may be used:

```
    dakota -i dakota.in -o dakota.out -e dakota.err
```

Alternatively, any of a variety of UNIX redirection variants can be used. The simplest of these redirects stdout to another file:

```
    dakota -i dakota.in > dakota.out
```

To append to a file rather than overwrite it, ">>" is used in place of ">". To redirect stderr as well as stdout, a "&" is appended with no embedded space, i.e. ">&" or ">>&" is used. To override the noclobber environment variable (if set) in order to allow overwriting of an existing output file or appending of a file that does not yet exist, a "`!`" is appended with no embedded space, i.e. "`>!`", "`>&!`", "`>>!`", or "`>>&!`" is used.

To run the dakota process in the background, append an ampersand symbol (&) to the command with an embedded space, e.g.:

```
    dakota -i dakota.in > dakota.out &
```

Refer to [1] for more information on UNIX redirection and background commands.

2.2 Rosenbrock and Textbook Test Problems

Many of the example problems in this chapter use the Rosenbrock function [32], which has the form:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{2.1}$$

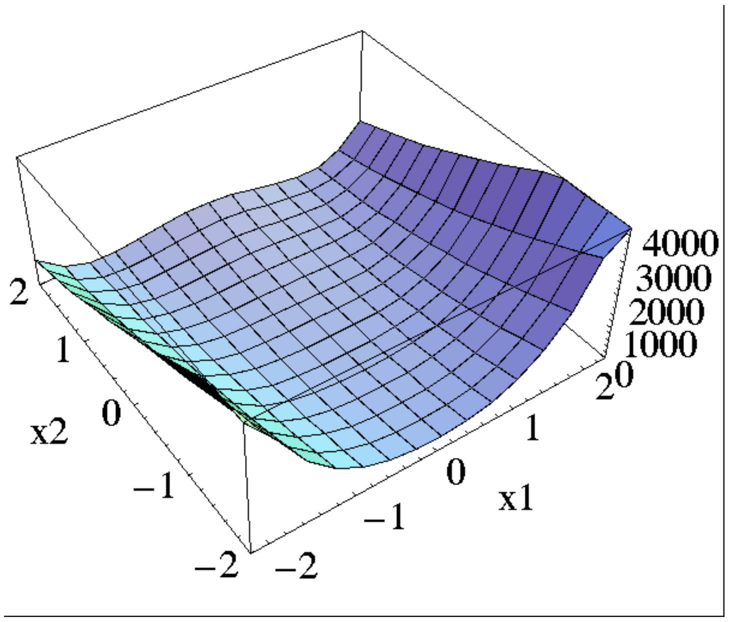


Figure 2.1: A 3-D plot of Rosenbrock's function.

A three-dimensional plot of this function is shown in Figure 2.1, where both x_1 and x_2 range in value from -2 to 2. Figure 2.2 shows a contour plot for Rosenbrock's function. An optimization problem using Rosenbrock's function is formulated as follows:

$$\begin{array}{rl} \texttt{minimize:} & f(x_1, x_2) \\ & \mathbf{x} \in \Re^2 \\ \texttt{subject to:} & -2 \le x_1 \le 2 \\ & -2 \le x_2 \le 2 \end{array} \tag{2.2}$$

Note that there are no linear or nonlinear constraints in this formulation, so this is a bound constrained optimization problem. The unique solution to this problem lies at the point $(x_1, x_2) = (1, 1)$ where the function value is zero.

The two-variable version of the "textbook" example problem provides a nonlinearly constrained optimization test case. It is formulated as:

```
minimize
```

$$f = (x_1 - 1)^4 + (x_2 - 1)^4 \tag{2.3}$$

```
subject to
```

$$g_1 = x_1^2 - \frac{x_2}{2} \le 0 \tag{2.4}$$

$$g_1 = x_2^2 - \frac{x_1}{2} \le 0 \tag{2.5}$$

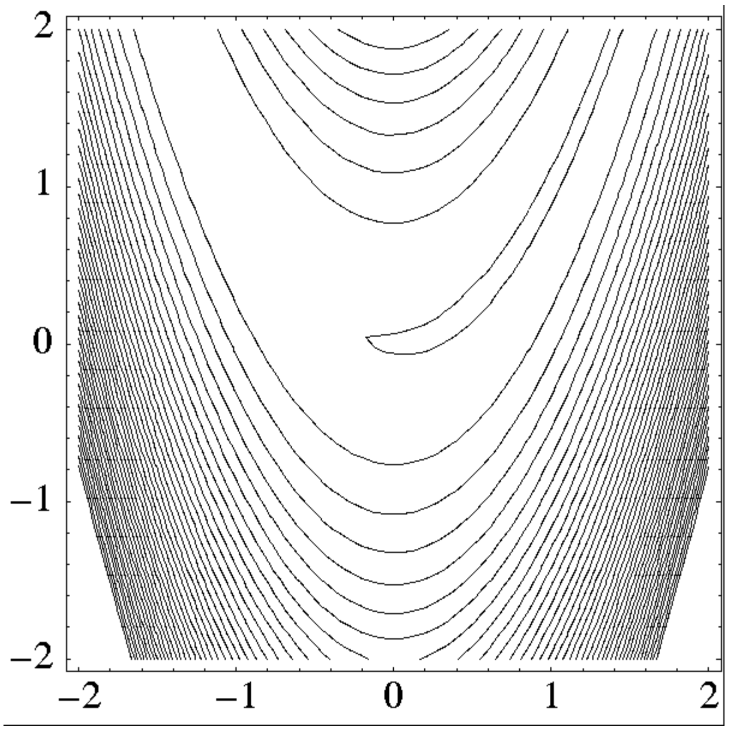


Figure 2.2: Contours of Rosenbrock's function with variable x_1 on the bottom axis.

$$-0.5 \leq x_1 \leq 5.8 \tag{2.6}$$

$$-2.9 \leq x_2 \leq 2.9 \tag{2.7}$$

Contours of this example problem are illustrated in Figure 2.3, with a close-up view of the feasible region given in Figure 2.4.

For the textbook example problem, the unconstrained minimum occurs at $(x_1, x_2) = (1, 1)$. However, the inclusion of the constraints moves the minimum to $(x_1, x_2) = (0.5, 0.5)$.

Several other example problems are available. See Chapter 20 for a description of these example problems as well as further discussion of the Rosenbrock and textbook example problems.

2.3 DAKOTA Input File Format

All of the DAKOTA input files for the simple example problems presented here are included in the distribution tar files within the directory `/Dakota/GettingStarted/Examples`. A simple DAKOTA input file for a two-dimensional parameter study on Rosenbrock's function is shown in Figure 2.5 (filename: `dakota_rosenbrock_2d.in`). This input file will be used to describe the basic format and syntax used in all DAKOTA input files.

There are five specification blocks that may appear in DAKOTA input files. These are identified in the input file using the following keywords: variables, interface, responses, method, and strategy. These keyword blocks can appear in any order in a DAKOTA input file. At least one *variables*, *interface*, *responses*, and *method* specification must appear, and no more than one *strategy* specification should appear. In Figure 2.5, one of each of the keyword blocks is used. Additional syntax features include the use of the

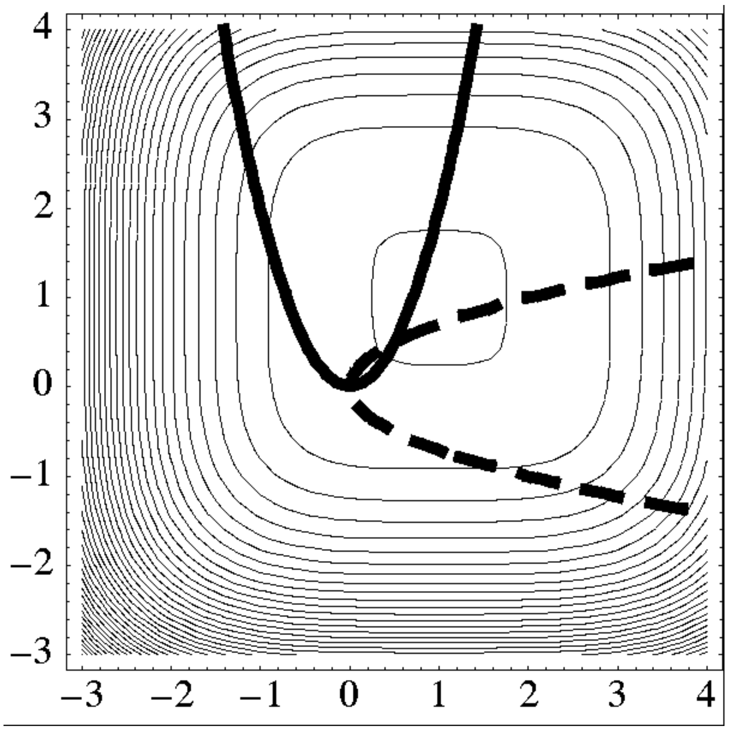


Figure 2.3: Contours of the textbook optimization problem showing constraints g_1 (solid) and g_2 (dashed). The feasible region lies at the intersection of the two constraints.

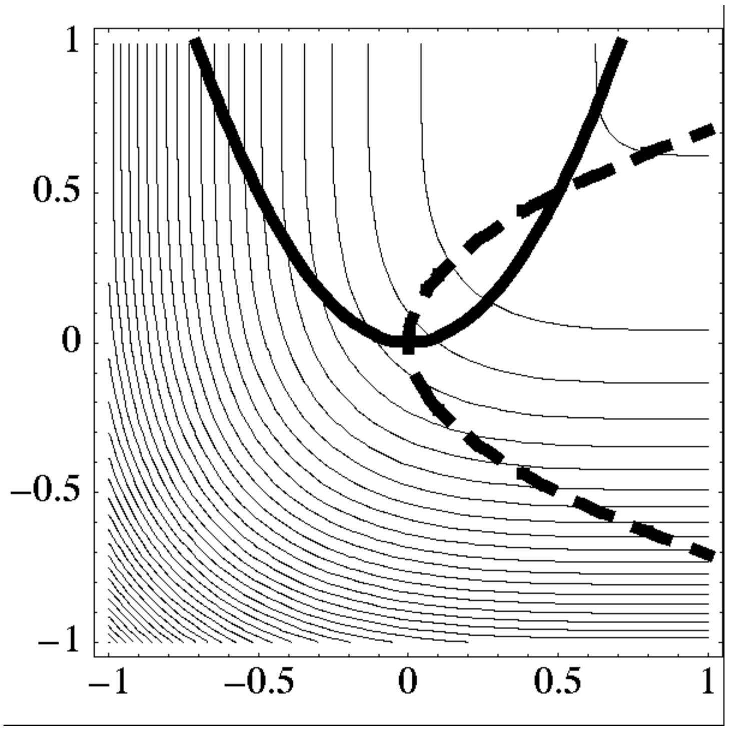


Figure 2.4: A close-up view of the feasible region for the textbook example problem. The constrained optimum point is at $(x_1, x_2) = (0.5, 0.5)$.

```
strategy,                                                   \
        single_method                                       \
          graphics                                          \
          tabular_graphics_data

method,                                                     \
        multidim_parameter_study                            \
          partitions = 8 8                                  \

model,                                                      \
        single

variables,                                                  \
        continuous_design = 2                               \
          cdv_lower_bounds    -2.0     -2.0                 \
          cdv_upper_bounds     2.0      2.0                 \
          cdv_descriptors      'x1'     'x2'                \

interface,                                                  \
        direct                                              \
          analysis_driver = 'rosenbrock'                    \

responses,                                                  \
        num_objective_functions = 1                         \
        no_gradients                                        \
        no_hessians
```

Figure 2.5: The DAKOTA input file for the 2-D parameter study example problem.

backslash symbol (\) to escape the newline character in order to split a keyword onto multiple lines for readability, use of the # symbol to indicate a comment, use of single quotes for string inputs (e.g., 'x1'), the use of commas and/or white space for separation of specifications, and the use of "=" symbols to optionally enhance the association of supplied data. See the DAKOTA Reference Manual [17] for additional details on this input file syntax.

The *variables* section of the input file specifies the characteristics of the parameters that will be used in the problem formulation. The variables can be continuous or discrete, and can be classified as design variables, uncertain variables, or state variables. See Chapter 4 for more information on the types of variables supported by DAKOTA. The *variables* section shown in Figure 2.5 specifies that there are two continuous design variables. The sub-specifications for continuous design variables use the abbreviation cdv in the input file and include the descriptors "x1" and "x2" as well as lower and upper bounds for these variables. The information about the variables is organized in column format for readability. So, both variables x_1 and x_2 have a lower bound of -2.0 and an upper bound of 2.0.

The *interface* section of the input file specifies what approach will be used to map variables into responses as well as details on how DAKOTA will pass data to and from a simulation code. In this example, a test function internal to DAKOTA is used, but the data may also be obtained from a simulation code that is external to DAKOTA. The keyword `application` indicates the use of an interface to an application code (as opposed to an `approximation` interface) and the keyword `direct` indicates the use of a function linked directly into DAKOTA. The `analysis_driver` keyword indicates the name of the test function. This is all that is needed since files will not be used to pass data between DAKOTA and the simulation code.

The *responses* section of the input file specifies the types of data that the interface will return to DAKOTA. For the example shown in Figure 2.5, there is only one objective function, as indicated by the keyword `num_objective_functions = 1`. Since there are no constraints associated with Rosenbrock's function, the keywords associated with constraint specifications are omitted. The keywords `no_gradients` and `no_hessians` indicate that gradient and Hessian data are not needed.

The *method* section of the input file specifies the iterative technique that DAKOTA will employ, such as a parameter study, optimization method, data sampling technique, etc. In Figure 2.5, the keyword `multidim_parameter_study` specifies a multidimensional parameter study, while the keyword partitions denotes the number of intervals per variable. In this case, there will be eight intervals (nine data points) evaluated between the lower and upper bounds of both variables (bounds provided previously in the *variables* section), for a total of 81 response function evaluations.

The final section of the input file shown in Figure 2.5 is the *strategy* section. This keyword section is used to specify some of DAKOTA's advanced meta-procedures such as multi-level optimization, surrogate-based optimization, branch-and-bound optimization, and optimization under uncertainty. See Chapter 13 for more information on these meta-procedures. The *strategy* section also contains the settings for DAKOTA's graphical output (via the `graphics` flag) and the tabular data output (via the `tabular_graphics_data` keyword).

2.4 Example Problems

2.4.1 Two-Dimensional Parameter Study

The 2-D parameter study example problem listed in Figure 2.5 is executed by DAKOTA using the following command:

```
    dakota -i dakota_rosenbrock_2d.in > 2d.out
```

The output of the DAKOTA run is directed to the file named `2d.out`. For comparison, the file `2d.out.sav` is included in the `/Dakota/GettingStarted/Examples` directory. As for many

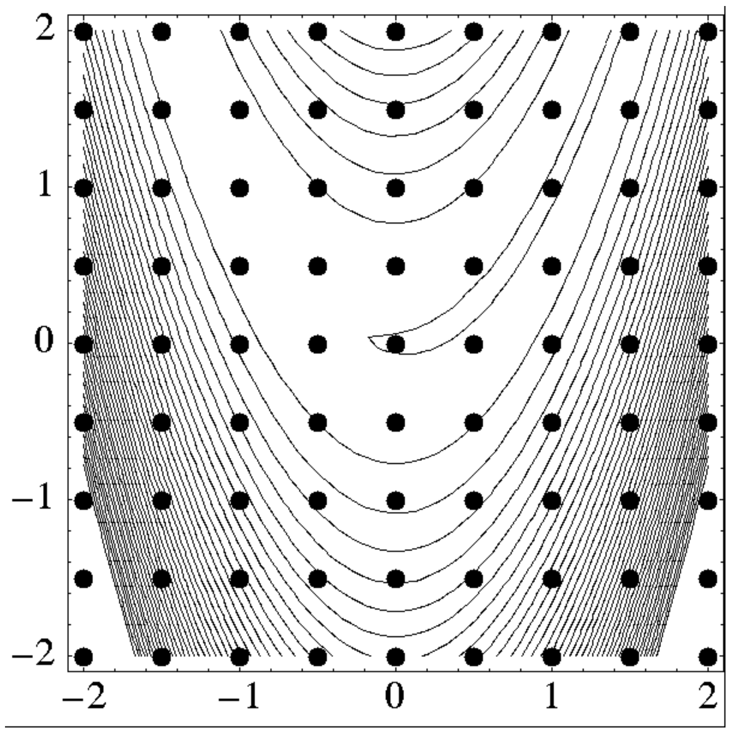


Figure 2.6: The dots indicate the location of the design points evaluated in the 2-D parameter study.

of the examples, DAKOTA provides a report on the best design point located during the study at the end of these output files.

This 2-D parameter study produces the grid of data samples shown in Figure 2.6. Note that the `graphics` flag in the *strategy* section of the input file has been commented out since, for this example, the iteration history plots created by DAKOTA are not particularly instructive. More interesting visualizations can be created by importing DAKOTA's tabular data into an external graphics/plotting package. Common graphics and plotting packages include Mathematica, Matlab, Microsoft Excel, Origin, Tecplot, and many others (Sandia National Laboratories and the DAKOTA developers do not endorse any of these commercial products).

2.4.2 Vector Parameter Study

In addition to the multidimensional parameter study, DAKOTA can perform a vector parameter study, i.e., a parameter study between any two design points in an *n*-dimensional parameter space.

An input file for the vector parameter study is shown in Figure 2.7. The primary differences between this input file and the previous input file are found in the *variables* and *method* sections. In the variables section, the keywords for the bounds are removed and replaced with the keyword `cdv_initial_point` that specifies the starting point for the parameter study. In the method section, the `vector_parameter_study` keyword is used. The `final_point` keyword indicates the stopping point for the parameter study, and `num_steps` specifies the number of steps taken between the initial and final points in the parameter study.

The vector parameter study example problem is executed using the command

```
dakota -i dakota_rosenbrock_vector.in > vector.out
```

Figure 2.8 shows the graphics output created by DAKOTA. For this study, the simple DAKOTA graphics are more useful for visualizing the results. Figure 2.9 shows the locations of the 11 sample points generated

```
strategy,                                                        \
        single_method                                            \
          graphics                                               \
          tabular_graphics_data

method,                                                          \
        vector_parameter_study                                   \
          final_point = 1.1  1.3                                 \
          num_steps = 10                                         \

model,                                                           \
        single

variables,                                                       \
        continuous_design = 2                                    \
          cdv_initial_point   -0.3      0.2                      \
          cdv_descriptors      'x1'     'x2'                     \

interface,                                                       \
        direct                                                   \
          analysis_driver = 'rosenbrock'                         \

responses,                                                       \
        num_objective_functions = 1                              \
        no_gradients                                             \
        no_hessians
```

Figure 2.7: The DAKOTA input file for the vector parameter study example problem.

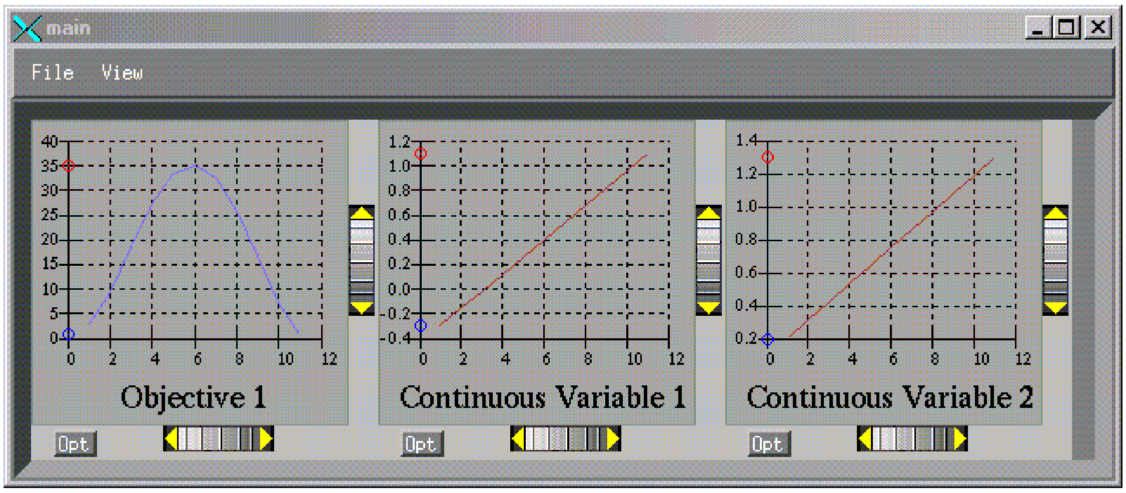


Figure 2.8: A screen capture of the DAKOTA graphics that are generated from the vector parameter study

in this study. It is evident from these figures that the parameter study starts within the banana-shaped valley, marches up the side of the hill, and then returns to the valley. The output file `vector.out.sav` is provided in the `/Dakota/GettingStarted/Examples` directory.

In addition to the vector and multidimensional examples shown, DAKOTA also supports list and centered parameter study methods. Refer to Chapter 8 for additional information.

2.4.3 Gradient-based Unconstrained Optimization

A DAKOTA input file for a gradient-based optimization of Rosenbrock's function is listed in Figure 2.10. The format of the input file is similar to that used for the parameter studies, but there are some new keywords in the responses and method sections. First, in the responses section of the input file, the keyword block starting with `numerical_gradients` specifies that a finite difference method will be used to compute gradients for the optimization algorithm. Note that the Rosenbrock function evaluation code inside DAKOTA has the capability to give analytical gradient values. To switch from finite difference gradient estimates to analytic gradients, uncomment the `analytic_gradients` keyword and comment out the four lines associated with the `numerical_gradients` specification. Next, in the method section of the input file, several new keywords have been added. In this section, the keyword `conmin_frcg` indicates the use of the Fletcher-Reeves conjugate gradient algorithm in the CONMIN optimization software package [65] for bound-constrained optimization. The keyword `max_iterations` is used to indicate the computational budget for this optimization (in this case, a single iteration includes multiple evaluations of Rosenbrock's function for the gradient computation steps and the line search steps). The keyword `convergence_tolerance` is used to specify one of CONMIN's convergence criteria (here, CONMIN terminates if the objective function value differs by less than the absolute value of the convergence tolerance for three successive iterations). And, finally, the `output` verbosity is set to `quiet`.

This DAKOTA input file is executed using the following command:

```
    dakota -i dakota_rosenbrock_grad_opt.in > grad_opt.out
```

A sample output file named `grad_opt.out.sav` is included in the `/Dakota/GettingStarted/Examples` directory. When this example problem is executed, DAKOTA creates some iteration history graphics similar to the screen capture shown in Figure 2.11. These plots show how the objective function and design parameters change in value during the optimization steps. The scaling of the horizontal and vertical axes can be changed by moving the scroll knobs on

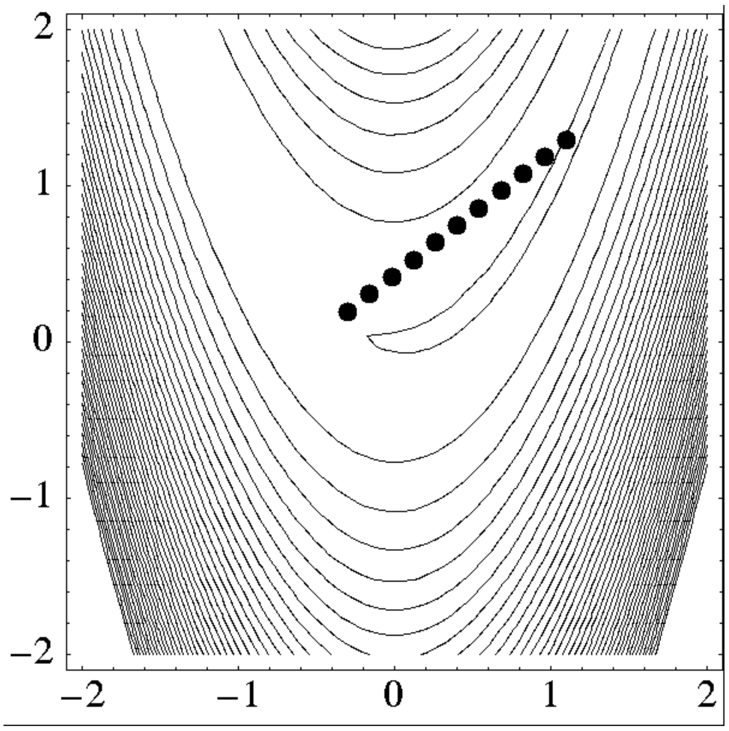


Figure 2.9: The dots indicate the location of the design points evaluated in the vector parameter study.

each plot. Also, the "Options" button allows the user to plot the vertical axes using a logarithmic scale. Note that log-scaling is only allowed if the values on the vertical axis are strictly greater than zero.

Figure 2.12 shows the iteration history of the optimization algorithm. The optimization starts at the point $(x_1, x_2) = (-1.2, 1.0)$ as given in the DAKOTA input file. Subsequent iterations follow the banana-shaped valley that curves around toward the minimum point at $(x_1, x_2) = (1.0, 1.0)$. Note that the function evaluations associated with the line search phase of each CONMIN iteration are not shown on the plot. At the end of the DAKOTA run, information is written to the output file to provide data on the optimal design point. This data includes the optimum design point parameter values, the optimum objective and constraint function values (if any), plus the number of function evaluations that occurred and the amount of time that elapsed during the optimization study.

2.4.4 Gradient-based Constrained Optimization

This example demonstrates the use of a gradient-based optimization algorithm on a nonlinearly constrained problem. The "textbook" example problem (see Section 2.2) is used for this purpose and the DAKOTA input file for this example problem is shown in Figure 2.13. This input file is similar to the input file for the unconstrained gradient-based optimization example problem involving the Rosenbrock function. Note the addition of commands in the responses section of the input file that identify the number and type of constraints, along with the upper bounds on these constraints. The commands `direct` and `analysis_driver = 'text_book'` specify that DAKOTA will execute its internal version of the textbook problem.

This example problem is executed by using the following command:

```
    dakota -i dakota_textbook.in > textbook.out
```

For comparison purposes, the file `textbook.out.sav` is included in

```
strategy,                                                \
        single_method                                    \
          graphics                                       \
          tabular_graphics_data

method,                                                  \
        conmin_frcg                                      \
          max_iterations = 100                           \
          convergence_tolerance = 1e-4                   \

model,                                                   \
        single

variables,                                               \
        continuous_design = 2                            \
          cdv_initial_point   -1.2      1.0              \
          cdv_lower_bounds    -2.0     -2.0              \
          cdv_upper_bounds     2.0      2.0              \
          cdv_descriptors      'x1'     'x2'             \

interface,                                               \
        direct                                           \
          analysis_driver = 'rosenbrock'                 \

responses,                                               \
        num_objective_functions = 1                      \
        numerical_gradients                              \
          method_source dakota                           \
          interval_type forward                          \
          fd_gradient_step_size = 1.e-5                  \
        no_hessians
```

Figure 2.10: The DAKOTA input file for the gradient-based optimization example problem.

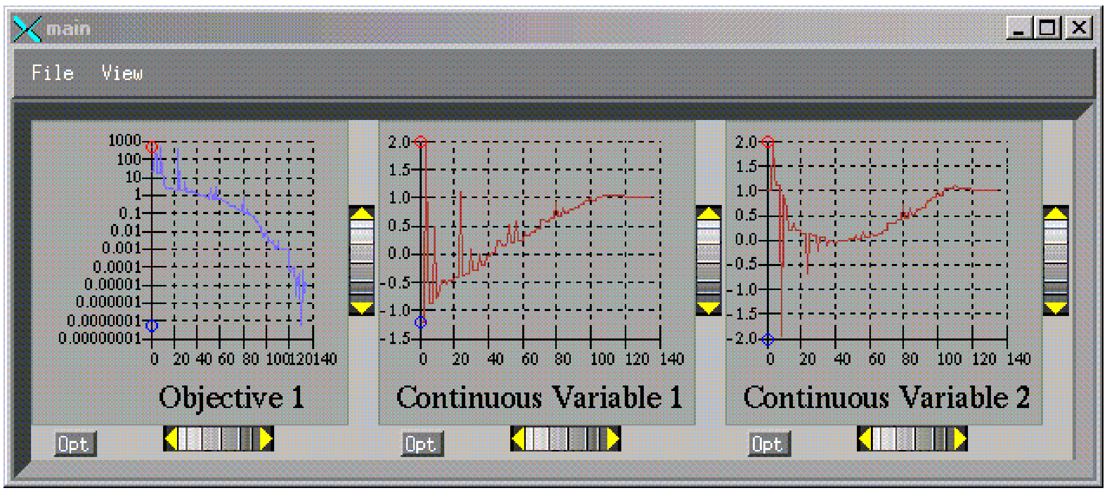


Figure 2.11: A screen capture of the DAKOTA output graphics showing the iteration history for the gradient-based optimization example.

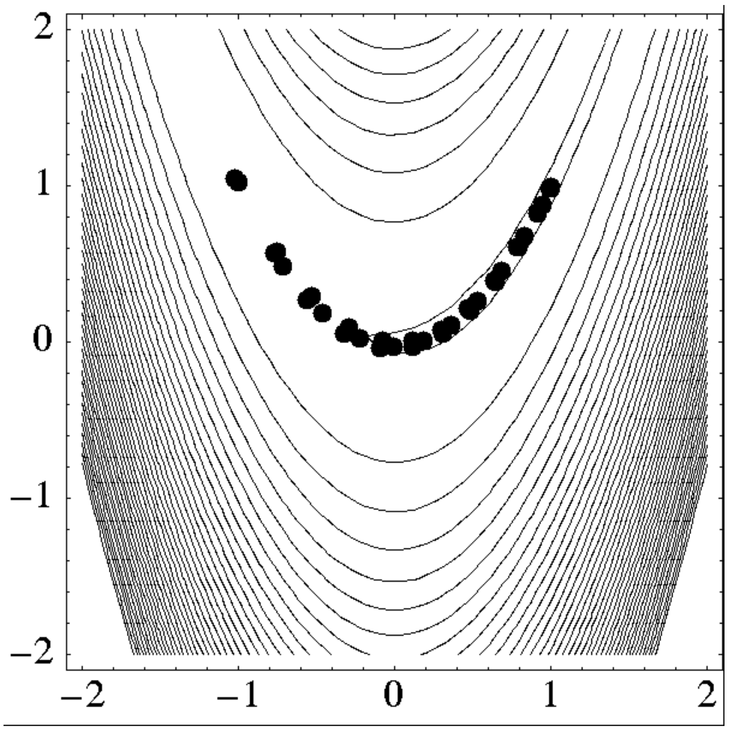


Figure 2.12: The sequence of design points evaluated during the gradient-based optimization of Rosenbrock's function (line search points omitted).

`/Dakota/GettingStarted/Examples`. The results of the optimization example problem are listed at the end of the `textbook.out` file. This information shows that the optimizer stopped at the point $(x_1, x_2) = (0.5, 0.5)$, where both constraints are satisfied, and where the objective function value is 0.125. This progress of the optimization algorithm is shown in Figure 2.14 where the dots correspond to end point of each iteration in the algorithm. The starting point is $(x_1, x_2) = (4.0, 0.0)$ where constraint g_1 is violated and constraint g_2 is satisfied. The optimizer takes a sequence of steps to minimize the objective function while reducing the infeasibility of g_1 and retaining the feasibility of g_2. The optimization graphics are also shown in Figure 2.15.

2.4.5 Nonlinear Least Squares Methods for Optimization

Both the Rosenbrock and textbook example problems can be formulated as least squares minimization problems (see Section 20.1 and Section 20.2). For example, the Rosenbrock problem can be cast as:

$$\texttt{minimize}\ (f_1)^2 + (f_2)^2 \tag{2.8}$$

where $f_1 = 10(x_2 - x_1^2)$ and $f_2 = (1 - x_1)$. When using a least squares approach to minimize a function, each of the least squares terms $f_1, f_2, \ldots$ is driven to zero. This formulation permits the use of specialized algorithms that can be more efficient than general purpose optimization algorithms. See Chapter 12 for more detail on the algorithms used for least squares minimization, as well as a discussion on the types of engineering design problems (e.g., parameter estimation) that can make use of the least squares approach.

Figure 2.16 is a listing of the DAKOTA input file `dakota_rosenbrock_ls.in`. This input file differs from the input file shown in Figure 2.10 in several key areas. The responses section of the input file uses the keyword `num_least_squares_terms = 2` instead of the `num_objective_functions = 1`. The keywords in the interface section show that the UNIX system call method is used to run the C++ anal-

```
strategy,                                                       \
        single_method

method,                                                         \
        dot_mmfd,                                               \
          max_iterations = 50,                                  \
          convergence_tolerance = 1e-4

variables,                                                      \
        continuous_design = 2                                   \
          cdv_initial_point    0.9    1.1                       \
          cdv_upper_bounds     5.8    2.9                       \
          cdv_lower_bounds     0.5   -2.9                       \
          cdv_descriptor       'x1'   'x2'

interface,                                                      \
        fork                                            \
          analysis_driver =       'text_book'                   \

responses,                                                      \
        num_objective_functions = 1                             \
        num_nonlinear_inequality_constraints = 2                \
        numerical_gradients                                     \
          method_source dakota                                  \
          interval_type central                                 \
          fd_gradient_step_size = 1.e-4                         \
        no_hessians
```

Figure 2.13: The DAKOTA input file for the nonlinearly constrained gradient-based optimization example problem.

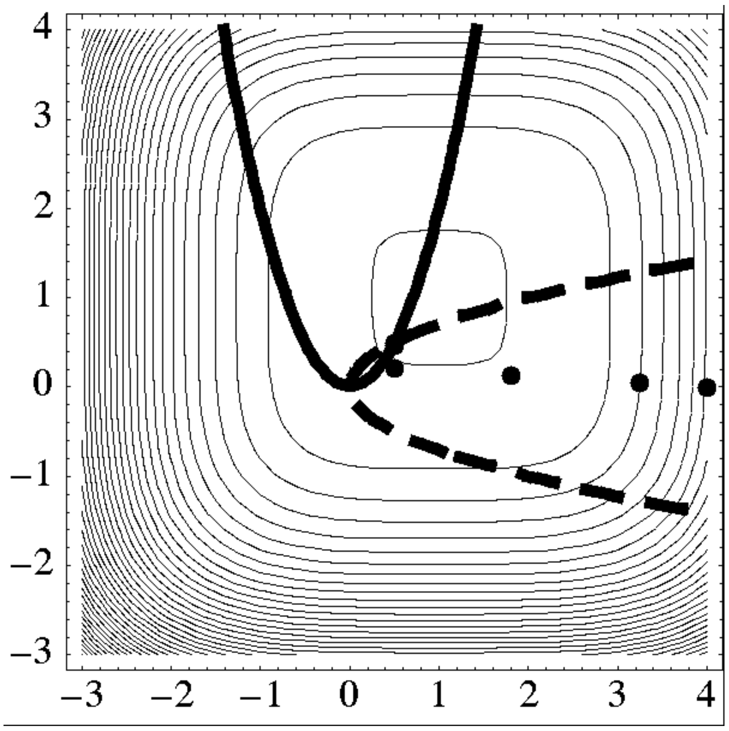


Figure 2.14: Iteration history of the textbook example problem (iterations marked by solid dots).

ysis code named `rosenbrock_ls`. The method section of the input file shows that the Gauss-Newton algorithm from the OPT++ library [50] (`optpp_g_newton`) is used in this example. For DAKOTA Version 3.1, the Gauss-Newton and NLSSOL SQP algorithms are available for exploiting the special mathematical structure of least squares minimization problems.

The input file listed in Figure 2.16 is executed using the command:

```
dakota -i dakota_rosenbrock_ls.in > leastsquares.out
```

The file `leastsquares.out.sav` is included in the directory `/Dakota/GettingStarted/Examples`. The optimization results at the end of this file show that the least squares minimization approach has found the same optimum design point, $(x1, x2) = (1.0, 1.0)$, as was found using the conventional gradient-based optimization approach. The iteration history of the least squares minimization is given in Figure 2.17, and shows that 90 function evaluations were needed for convergence. In this example the least squares approach required about the same number of function evaluations as did conventional gradient-based optimization. However, in many cases the least squares algorithm will converge more rapidly in the vicinity of the solution.

2.4.6 Nongradient-based Optimization via Pattern Search

In addition to gradient-based optimization algorithms, DAKOTA also contains a variety of nongradient-based algorithms. One particular nongradient-based algorithm for local optimization is known as pattern search (see Chapter 1 for a discussion of local versus global optimization). The DAKOTA input file shown in Figure 2.18 applies a pattern search method to minimize the Rosenbrock function. While this provides for an interesting comparison to the previous example problems in this chapter, the Rosenbrock function is not the best test case for a pattern search method. That is, pattern search methods are better suited to problems where the gradients are too expensive to evaluate, inaccurate, or nonexistent; situations common

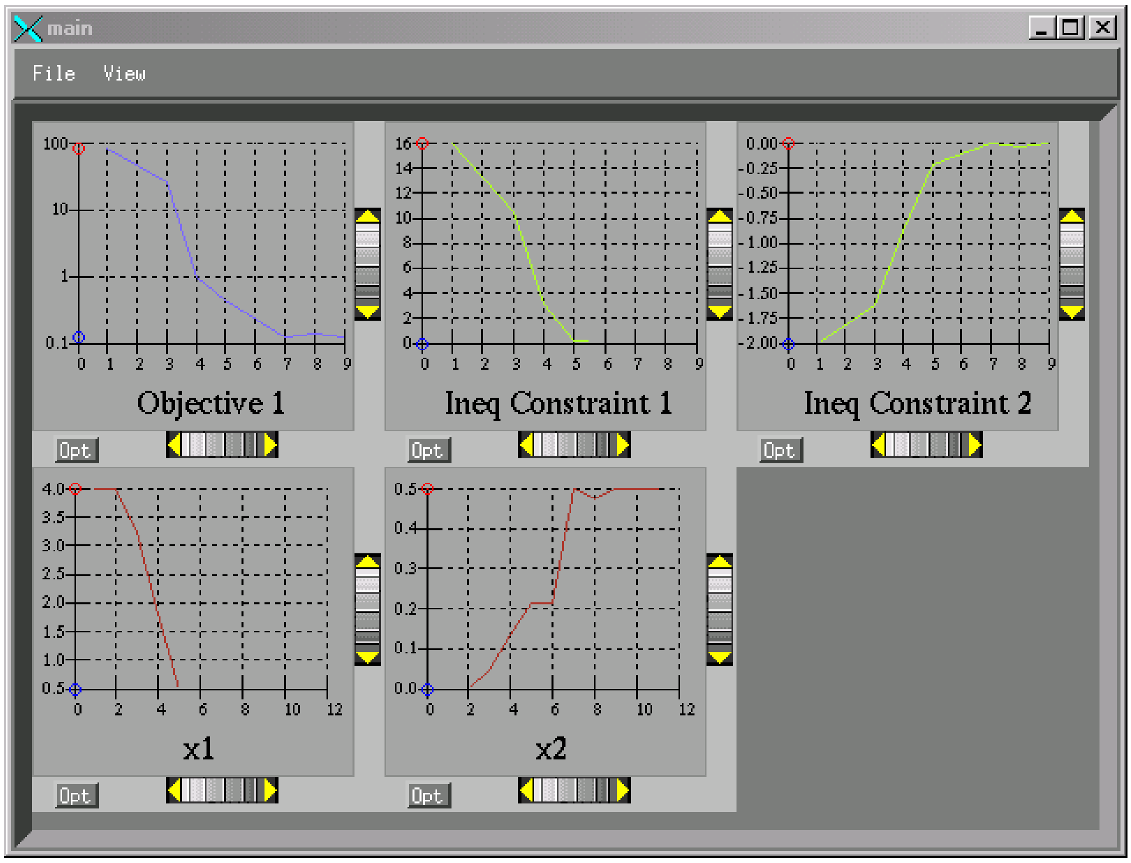


Figure 2.15: The iteration history of the textbook example problem shows how the objective function was reduced during the search for a feasible design point.

```
strategy,                                                   \
        single_method                                       \
          graphics                                          \
          tabular_graphics_data

method,                                                     \
        optpp_g_newton                                      \
          max_iterations = 100                              \
          convergence_tolerance = 1e-4                      \

model,                                                      \
        single

variables,                                                  \
        continuous_design = 2                               \
          cdv_initial_point   -1.2      1.0                 \
          cdv_lower_bounds    -2.0     -2.0                 \
          cdv_upper_bounds     2.0      2.0                 \
          cdv_descriptors      'x1'     'x2'                \

interface,                                                  \
        direct                                              \
          analysis_driver = 'rosenbrock'                    \

responses,                                                  \
        num_least_squares_terms = 2                         \
        analytic_gradients                                  \
        no_hessians
```

Figure 2.16: DAKOTA input file for minimizing the Rosenbrock function using a least squares formulation.

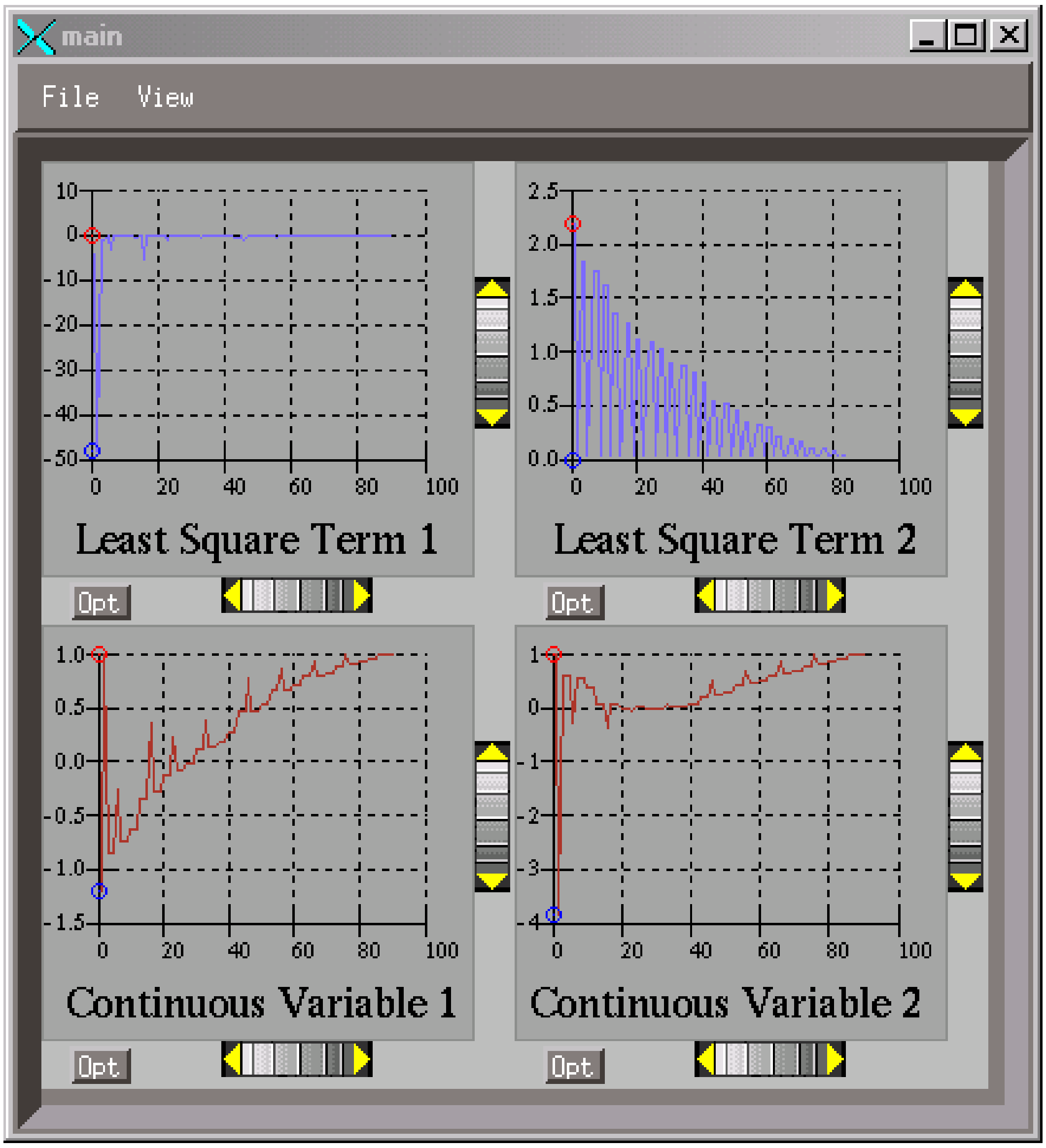


Figure 2.17: The iteration history for least squares terms f_1 and f_2 when minimizing the Rosenbrock function.

```
strategy,                                                        \
        single_method                                            \
          graphics                                               \
          tabular_graphics_data

method,                                                          \
        sgopt_pattern_search                                     \
          max_iterations = 1000                                  \
          max_function_evaluations = 2000                        \
          solution_accuracy = 1e-4                               \
          initial_delta = 0.05                                   \
          threshold_delta = 1e-8                                 \
          exploratory_moves best_all                             \
          contraction_factor = 0.75                              \

model,                                                           \
        single

variables,                                                       \
        continuous_design = 2                                    \
          cdv_initial_point    0.0      0.0                      \
          cdv_lower_bounds    -2.0     -2.0                      \
          cdv_upper_bounds     2.0      2.0                      \
          cdv_descriptors      'x1'     'x2'                     \

interface,                                                       \
        direct                                                   \
          analysis_driver = 'rosenbrock'                         \

responses,                                                       \
        num_objective_functions = 1                              \
        no_gradients                                             \
        no_hessians
```

Figure 2.18: A DAKOTA input file for a nongradient-based optimization example.

among many engineering optimization problems. It also should be noted that nongradient-based algorithms generally are applicable only to unconstrained or bound-constrained optimization problems, although the inclusion of general linear and nonlinear constraints in nongradient-based algorithms is an active area of research in the optimization community. For most users who wish to use nongradient-based algorithms on constrained optimization problems, the easiest route is to create a penalty function, i.e., a composite function that contains the objective function and the constraints, external to DAKOTA and then optimize on this penalty function. Most optimization textbooks will provide guidance on selecting and using penalty functions.

This DAKOTA input file shown in Figure 2.18 is similar to the input file for the gradient-based optimization, except it has a different set of keywords in the method section of the input file and the gradient specification in the responses section has been changed to `no_gradients`. The pattern search optimization algorithm used is part of the COLINY library [42]. See the DAKOTA Reference Manual [17] for more information on the *methods* section commands that can be used with COLINY algorithms.

This DAKOTA input file is executed using the following command:

```
    dakota -i dakota_rosenbrock_ps_opt.in > ps_opt.out
```

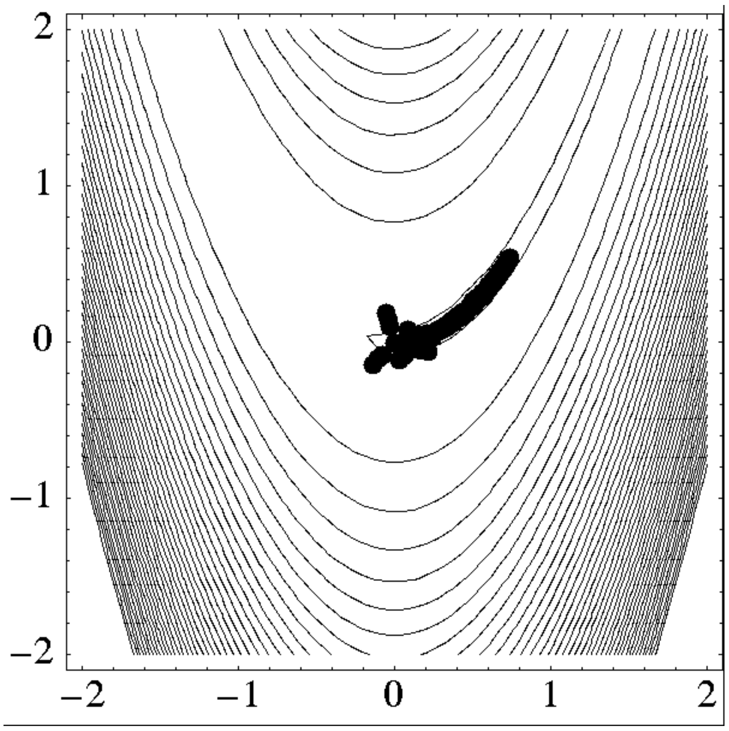


Figure 2.19: The sequence of design points evaluated during a nongradient-based pattern search optimization of Rosenbrock's function.

The file `ps_opt.out.sav` is included in the `/Dakota/GettingStarted/Examples` directory. For this run, the optimizer was given an initial design point of $(x_1, x_2) = (0.0, 0.0)$ and was limited to 2000 function evaluations. In this case, the pattern search algorithm stopped short of the optimum at $(x_1, x_2) = (1.0, 1, 0)$, although it was making progress in that direction when it was terminated (eventually, it would have reached the minimum point).

The iteration history is provided in Figure 2.19 which shows the locations of the function evaluations used in the pattern search algorithm. Figure 2.20 provides a close-up view of the pattern search function evaluations used at the start of the algorithm. The simplex pattern is clearly visible at the start of the iteration history, and the decreasing size of the simplex pattern is evident at the design points move toward $(x_1, x_2) = (1.0, 1.0)$.

While pattern search algorithms are useful in many optimization problems, this example shows some of the drawbacks to this algorithm. While a pattern search method may make good initial progress towards an optimum, it is often slow to converge. On a smooth, differentiable function such as Rosenbrock's function, a nongradient-based method will not be as efficient as a gradient-based method. However, there are many engineering design applications where gradient information is inaccurate or unavailable, which renders gradient-based optimizers ineffective. Thus, pattern search algorithms (and other nongradient-based algorithms such as genetic algorithms and simulated annealing) are often good choices in complex engineering applications when the quality of gradient data is suspect.

2.4.7 Nongradient-based Optimization via Genetic Algorithm

In contrast to pattern search algorithms, which are local optimization methods, genetic algorithms (GA) are global optimization methods. As was described above for the pattern search algorithm, the Rosenbrock function is not an ideal test problem for showcasing the capabilities of genetic algorithms. Rather, GAs are best suited to optimization problems that have multiple local optima, and where gradients are either too

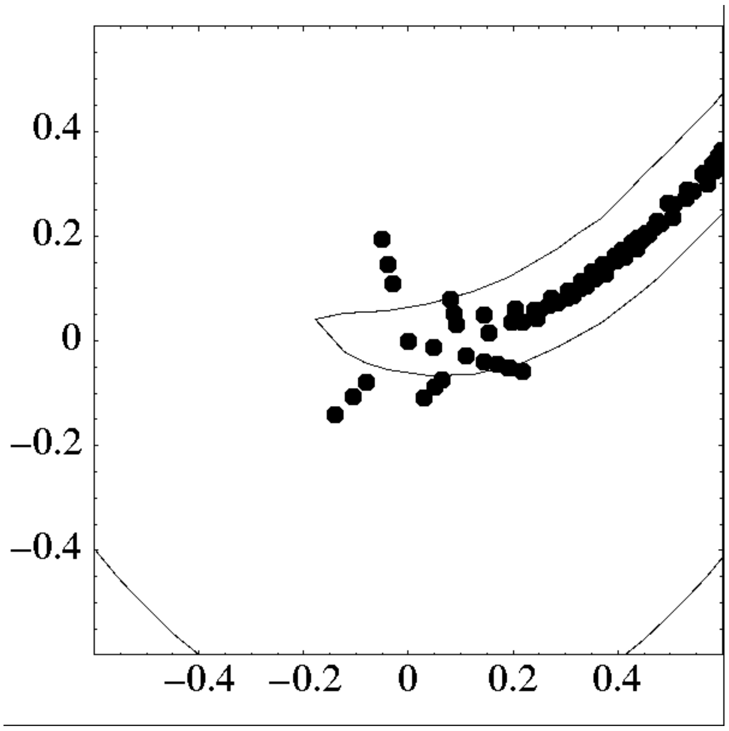


Figure 2.20: A close-up view shows the shape of the simplex pattern used at the start of the pattern search algorithm.

expensive to compute or do not exist.

Genetic algorithms, also known as Evolutionary Algorithms (EAs), are based on Darwin's theory of survival of the fittest. The GA algorithm starts with a randomly selected population of design points in the parameter space, where the values of the design parameters form a "genetic string," which is analogous to DNA in a biological system, that uniquely represents each design point in the population. The GA then follows a sequence of generations, where the best design points in the population (i.e., those having low objective function values) are considered to be the most "fit" and are allowed to survive and reproduce. The GA simulates the evolutionary process by employing the mathematical analogs of processes such as natural selection, breeding, and mutation. Ultimately, the GA identifies a design point, or a family of design points, that minimize the objective function of the optimization problem. An extensive discussion of GAs is beyond the scope of this text, but may be found in a variety of sources (cf., [40] pp. 149-158; [37]). Detailed information on the GA algorithms available in DAKOTA is given in the DAKOTA Reference Manual [17]. The COLINY library, which provides the GA software that has been linked into DAKOTA, is described in Reference [42].

Figure 2.21 shows a DAKOTA input file that uses a genetic algorithm to minimize the Rosenbrock function. For this example the GA has a population size of 50. At the start of the first generation, a random number generator is used to select 50 design points that will comprise the initial population. *[A specific seed value is used in this example to generate repeatable results, although, in general, one should use the default setting which allows the GA to choose a random seed.]* A two-point crossover technique is used to exchange genetic string values between the members of the population during the GA breeding process. The result of the breeding process is a population comprised of the 10 best "parent" design points (elitist strategy) plus 40 new "child" design points. The GA optimization process will be terminated after either 6,000 iterations (generations of the GA) or 10,000 function evaluations. The GA software available in DAKOTA provides the user with much flexibility in choosing the settings used in the optimization process. See [17] and [42] for details on these settings.

```
strategy,                                               \
        single_method                                   \
          graphics                                      \
          tabular_graphics_data

method,                                                 \
        sgopt_pga_real                                  \
          max_iterations = 6000                         \
          max_function_evaluations = 10000              \
          seed = 11011011                               \
          population_size = 50                          \
          replacement_type elitist = 10                 \
          crossover_type two_point                      \

model,                                                  \
        single

variables,                                              \
        continuous_design = 2                           \
          cdv_lower_bounds    -2.0     -2.0             \
          cdv_upper_bounds     2.0      2.0             \
          cdv_descriptors      'x1'     'x2'            \

interface,                                              \
        direct                                          \
          analysis_driver = 'rosenbrock'                \

responses,                                              \
        num_objective_functions = 1                     \
        no_gradients                                    \
        no_hessians
```

Figure 2.21: A DAKOTA input file that specifies the use of a genetic algorithm for optimizing Rosenbrock's function.

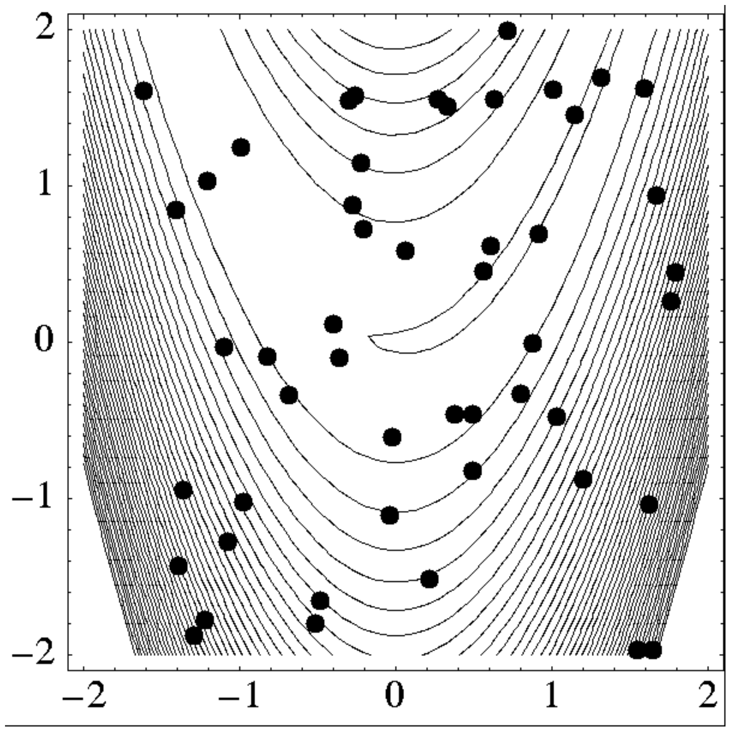


Figure 2.22: The 50 design points in the initial population selected by the genetic algorithm.

The input file is executed by DAKOTA using the following command:

```
dakota -i dakota_rosenbrock_ga_opt.in >! ga_opt.out
```

where the file `ga_opt.out.sav` has been included in `/Dakota/GettingStarted/Examples`. The GA optimization results printed at the end of this file show that the best design point found was $(x_1, x_2) = (0.96, 0.93)$. The file `ga_tabular.dat.sav` provides a listing of the design parameter values and objective function values for all 10,000 design points evaluated during the running of the GA. Figure 2.22 shows the population of 50 randomly selected design points that comprise the first generation of the GA, and Figure 2.23 shows the final population of 50 design points, where most of the 50 points are clustered near $(x_1, x_2) = (0.96, 0.93)$.

As described above, a GA is not well-suited to an optimization problem involving a smooth, differentiable objective such as the Rosenbrock function. Rather, GAs are better suited to optimization problems where conventional gradient-based optimization fails, such as situations where there are multiple local optima and/or gradients cannot be computed. In such cases, the computational expense of a GA is warranted since other optimization methods are not applicable or impractical. In many optimization problems, GAs often quickly identify promising regions of the design space where the global minimum may be located. However, a GA can be slow to converge to the optimum. For this reason, it can be an effective approach to combine the global search capabilities of a GA with the efficient local search of a gradient-based algorithm in a *multilevel hybrid optimization* strategy. In this approach, the optimization starts by using a few iterations of a GA to provide the initial search for a good region of the parameter space (low objective function and/or feasible constraints), and then it switches to a gradient-based algorithm (using the best design point found by the GA as its starting point) to perform an efficient local search for an optimum design point. More information on this multilevel hybrid approach is provided in Chapter 13.

In addition to the genetic algorithm capabilities in the `coliny_ea` method, there is a single-objective genetic algorithm method called `soga`. The major differences are that `soga` allows a warm start (e.g. you can read in starting solutions from a file), and it allows one to specify a mix of continuous and discrete

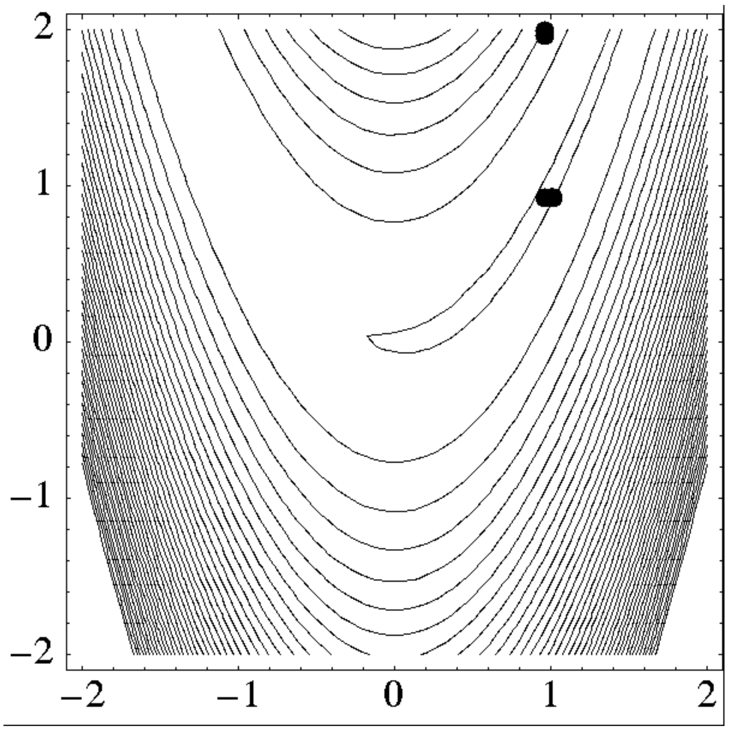


Figure 2.23: The 50 design points in the final population selected by the genetic algorithm. Most of the points are clustered near $(x_1, x_2) = (0.96, 0.93)$.

design variables. For more information on soga, see Chapter 11.

2.4.8 Multiobjective Optimization

Multiobjective optimization means that there are two or more objective functions that you wish to optimize simultaneously. Often these are conflicting objectives, such as cost and performance. The answer to a multi-objective problem is usually not a single point. Rather, it is a set of points called the Pareto front. Each point on the Pareto front satisfies the Pareto optimality criterion, which is stated as follows: a feasible vector X^* is Pareto optimal if there exists no other feasible vector X which would improve some objective without causing a simultaneous worsening in at least one other objective. Thus, if a feasible point X' exists that CAN be improved on one or more objectives without worsening of another, it is not Pareto optimal: it is said to be “dominated” and the points along the Pareto front are said to be “non-dominated”.

Often multi-objective problems are addressed by simply assigning weights to the individual objectives, summing the weighted objectives, and turning the problem into a single-objective one which can be solved with a variety of optimization techniques. While this approach provides a useful “first cut” analysis (and is supported within DAKOTA, see Section 11.3), this approach has many limitations. The major limitation is that a linear weighted sum objective will not find optimal solutions if the true Pareto front is nonconvex. Also, if one wants to understand the effects of changing weights, this method can be computationally expensive. Since each optimization of a single weighted objective will find only one point near or on the Pareto front, many optimizations must be performed to get a good parametric understanding of the influence of the weights and to achieve a good sampling of the entire Pareto frontier.

With version 3.2 of DAKOTA, we have added a capability to perform multi-objective optimization based on a genetic algorithm method. This method is called `moga`. It is based on the idea that as the population evolves in a GA, solutions which are non-dominated are chosen to remain in the population. Until version

4.0 of DAKOTA, there was a selection_type choice of domination_count which performed a custom fitness assessment and selection operation together. As of version 4.0 of DAKOTA, that functionality has been broken into separate, more generally usable fitness assessment and selection operators called the domination count fitness assessor and below limit selector respectively. The effect of using these two operators is the same as the previous behavior of the domination_count selector. This means of selection works especially well on multi-objective problems because it has been specifically designed to avoid problems with aggregating and scaling objective function values and transforming them into a single objective. Instead, the fitness assessor works by ranking population members such that their resulting fitness is a function of the number of other designs that dominate them. The below_limit selector then chooses designs by considering the fitness of each. If the fitness of a design is above a certain limit, which in this case corresponds to a design being dominated by more than a specified number of other designs, then it is discarded. Otherwise it is kept and selected to go to the next generation. The one catch is that this selector will require that a minimum number of selections take place. `shrinkage_percentage` defines the minimum amount of selections that will take place if enough designs are available. It is interpreted as a percentage of the population size that must go on to the subsequent generation. To enforce this, the below_limit selector makes all the selections it would make anyway and if that is not enough, it relaxes its limit and makes selections from the remaining designs. It continues to do this until it has made enough selections. The moga method has many other important features. Complete descriptions can be found in the DAKOTA Reference Manual [17].

Figure 2.24 shows an example input file, `dakota_mogatest1.in`, which demonstrates some of the multi-objective capabilities available with the moga method. This example has three input variables and two objectives or response functions. Note that this method is referring to a different problem than the Rosenbrock function because we wanted to demonstrate the capability on a problem with two conflicting objectives. This example is taken from a testbed of multi-objective problems [75]. The final results from moga are output to a file called `finaldata.dat` in the directory in which you are running. This `finaldata.dat` file is simply a list of inputs and outputs. Plotting the output columns against each other allows one to see the Pareto front generated by `moga`. Figure 2.25 shows an example of the Pareto front based on the results of executing the file with the following command: `dakota -i dakota_mogatest1.in`. Note that a Pareto front easily shows the tradeoffs between Pareto optimal solutions. For example, look at the point with f1 and f2 values equal to (0.9, 0.25). One cannot improve (minimize) the value of objective function f1 without increasing the value of f2: another point on the Pareto front, (0.6, 0.6) represents a better value of objective f1 but a worse value of objective f2.

Sections 11.2 and 11.3 provide more information on multiobjective optimization. There are three detailed examples provided in Section 20.6.

2.4.9 Monte Carlo Sampling

Figure 2.26 shows the DAKOTA input file for an example problem which demonstrates some of the random sampling capabilities available in DAKOTA. In this example, the design parameters, x1 and x2, will be treated as uncertain parameters that have uniform distributions over the interval [-2, 2]. This is specified in the variables section of the input file, beginning with the keyword `uniform_uncertain`. For comparison, the keywords from the previous examples are retained, but have been commented out. Another change in the input file occurs in the responses section where the keyword `num_response_functions` is used in place of `num_objective_functions`. The final changes to the input file occur in the method section, where the keyword `nond_sampling` (nond is an abbreviation for nondeterministic) is used. The other keywords in the methods section of the input file specify the number of samples (200), the seed for the random number generator (17), the sampling method (random), and the response threshold (100.0). The `seed` specification allows a user to obtain repeatable results from multiple runs. If a seed value is not specified, then DAKOTA's sampling methods are designed to generate nonrepeatable behavior (by initializing the seed using a system clock). The keyword `response_thresholds` allows the user to specify threshold values for which DAKOTA will compute statistics on the response function output. Note that a unique threshold value can be specified for each response function.

```
strategy,                                                       \
        single_method                                           \

method,                                                         \
        moga                                                    \
        output silent                                           \
        seed = 10983                                            \
        max_function_evaluations = 2500                         \
        initialization_type                                     \
                unique_random                                   \
        crossover_type                                          \
                multi_point_parameterized_binary = 3            \
                        crossover_rate = 0.8                    \
        mutation_type                                           \
                replace_uniform                                 \
                        mutation_rate = 0.1                     \
        fitness_type                                            \
                domination_count                                \
        replacement_type                                        \
                below_limit = 6                                 \
                        shrinkage_percentage = 0.9              \
        convergence_type                                        \
                metric_tracker                                  \
                        percent_change = 0.05                   \
                        num_generations = 10

variables,                                              \
        continuous_design = 3                           \
          cdv_initial_point     0        0       0      \
          cdv_upper_bounds      4        4       4      \
          cdv_lower_bounds     -4       -4      -4      \
          cdv_descriptor       'x1'     'x2'    'x3'

interface,                                              \
        system                                          \
          analysis_driver = 'mogatest1'

responses,                                              \
        num_objective_functions = 2                     \
        no_gradients                                    \
        no_hessians
```

Figure 2.24: A DAKOTA input file that specifies the use of a multiple objective genetic algorithm (MOGA)

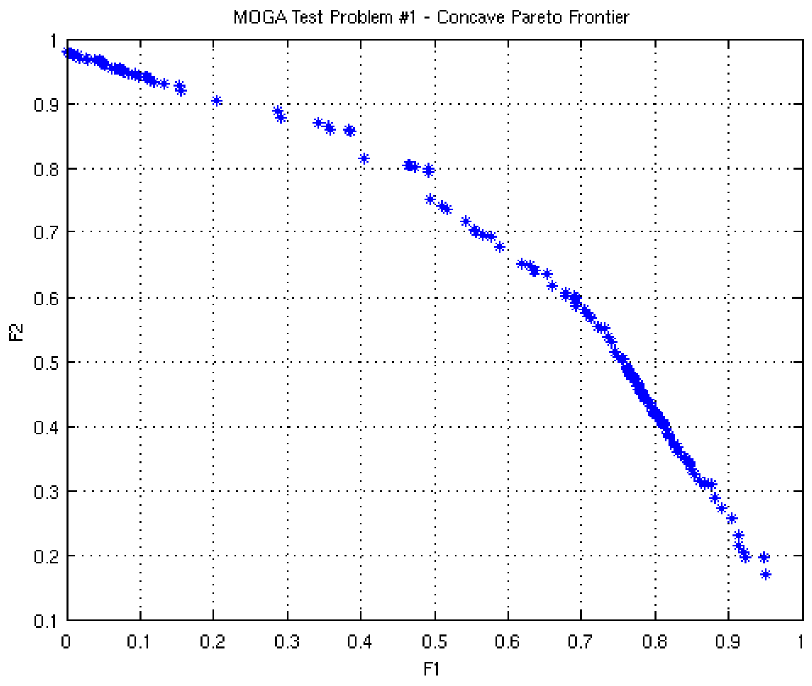


Figure 2.25: Pareto Front showing Tradeoffs between Function F1 and Function F2

In this example, DAKOTA will select 200 design points from within the parameter space, evaluate the value of Rosenbrock's function at all 200 points, and then perform some basic statistical calculations on the 200 response values.

This DAKOTA input file is executed using the following command:

```
dakota -i dakota_rosenbrock_nond.in > nond.out
```

See the file `nond.out.sav` in `/Dakota/GettingStarted/Examples` for comparison to the results produced by DAKOTA. Note that your results will differ from those in this file if your `seed` value differs or if no `seed` is specified.

The statistical data on the 200 Monte Carlo samples is printed at the end of the output file in the section that starts with "Statistics for each response function...." In this section, DAKOTA outputs the mean, standard deviation, coefficient of variation, and 95% confidence intervals for each of the response functions, followed by the percentages of the response function values that are above and below the response threshold values specified in the input file. Figure 2.27 shows the locations of the 200 sample sites within the parameter space of the Rosenbrock function.

2.4.10 Optimization with a User-Supplied Simulation Code - Case 1

Many of the previous examples made use of the direct interface to access the Rosenbrock and textbook test functions that are compiled into DAKOTA. In engineering applications, it is much more common to use the `system` or `fork` interface approaches within DAKOTA to manage external simulation codes. In both of these cases, the communication between DAKOTA and the external code is conducted through the reading and writing of short text files. For this example, the C++ program `rosenbrock.C` in `/Dakota/test` is used as the simulation code. This file is compiled to create the stand-alone `rosenbrock` executable that is referenced as the `analysis_driver` in Figure 2.28. This stand-alone program performs the same

```
strategy,                                            \
        single_method                                \
          graphics                                   \
          tabular_graphics_data

method,                                              \
        nond_sampling                                \
          samples = 200 seed = 17                    \
          sample_type random                         \
          response_levels = 100.0

model,                                               \
        single

variables,                                           \
        uniform_uncertain = 2                        \
          uuv_lower_bounds -2.0  -2.0        \
          uuv_upper_bounds  2.0   2.0        \
          uuv_descriptor         'x1'  'x2'

interface,                                           \
        direct                                       \
          analysis_driver = 'rosenbrock'             \

responses,                                           \
        num_response_functions = 1                   \
        no_gradients                                 \
        no_hessians
```

Figure 2.26: The DAKOTA input file for the Monte Carlo sampling example problem.

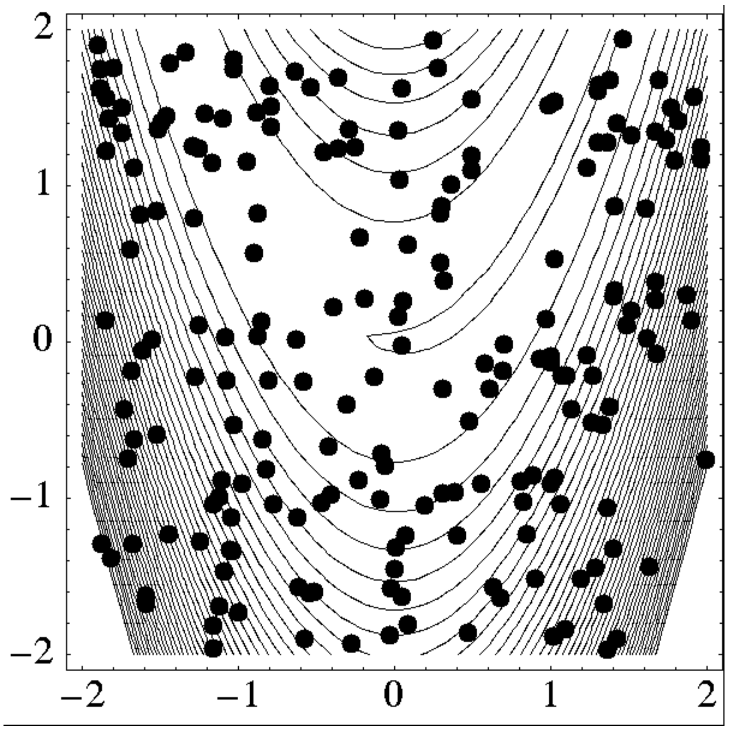


Figure 2.27: Locations in the parameter space of the 200 Monte Carlo samples using a uniform distribution for both x_1 and x_2.

function evaluations as DAKOTA's internal Rosenbrock test function.

Figure 2.28 shows the text of the DAKOTA input file named `dakota_rosenbrock_syscall.in` that is provided in the directory `/Dakota/GettingStarted/Examples`. The only differences between this input file and the one in Figure 2.10 occur in the *interface* keyword section. The keyword `system` indicates that DAKOTA will use system calls to create separate UNIX processes for executions of the user-supplied simulation code. The name of the simulation code, and the names for DAKOTA's parameters and results file are specified using the `analysis_driver`, `parameters_file`, and `results_file` keywords, respectively.

This example problem is executed using the command:

```
dakota -i dakota_rosenbrock_syscall.in > syscall.out
```

This run of DAKOTA takes longer to complete than the previous gradient-based optimization example since the `system` interface method has additional process creation and file I/O overhead, as compared to the internal communication that occurs when the `direct` interface method is used. The file `syscall.out.sav` is provided in the `/Dakota/GettingStarted/Examples` directory for comparison to the output results produced when executing the command given above.

To gain a better understanding of what exactly DAKOTA is doing with the `system` interface method, edit the input file to remove the comment symbols that are in front of the keywords `file_tag` and `file_save` and re-run DAKOTA. Check the listing of the local directory and you will see many new files with names such as `params.in.1`, `params.in.2`, etc., and `results.out.1`, `results.out.2`, etc. There is one `params.in.X` file and one `results.out.X` file for each of the function evaluations performed by DAKOTA. This is the file listing for `params.in.1`:

```
2 variables 1 functions
-1.2000000000e+00 x1
```

```
strategy,                                                \
        single_method                                    \
          graphics                                       \
          tabular_graphics_data

method,                                                  \
        conmin_frcg                                      \
          max_iterations = 100                           \
          convergence_tolerance = 1e-4                   \

model,                                                   \
        single

variables,                                               \
        continuous_design = 2                            \
          cdv_initial_point   -1.2      1.0              \
          cdv_lower_bounds    -2.0     -2.0              \
          cdv_upper_bounds     2.0      2.0              \
          cdv_descriptors      'x1'     'x2'             \

interface,                                               \
        system                                           \
          analysis_driver = 'rosenbrock'                 \
          parameters_file = 'params.in'                  \
          results_file    = 'results.out'

responses,                                               \
        num_objective_functions = 1                      \
        numerical_gradients                              \
          method_source dakota                           \
          interval_type forward                          \
          fd_gradient_step_size = 1.e-5                  \
        no_hessians
```

Figure 2.28: DAKOTA input file for gradient-based optimization using the system call interface to an external rosenbrock simulator.

```
    1.0000000000e+00 x2
    1 ASV_1
```

The first line gives the number of variables and the number of response functions. For optimization on Rosenbrock's function, there are two variables (x_1 and x_2) and one function (the objective function). The values of the variables are listed next in the file, with the descriptor tag ('x1' or 'x2' from the DAKOTA input file) following the numerical value. The last line of the parameters file is the syntax for DAKOTA's active set vector (ASV). There is one ASV line printed in the parameters file for each response function. In this case, the ASV value of 1 indicates that DAKOTA is requesting that the simulation code return the response function value to the file `results.out.X`. (ASV syntax: 1 = value of response function, 2 = gradient of response function, 4 = Hessian of response function, and any combination up to 7 = value, gradient, and Hessian of the response function. See Section 4.7 for more detail.)

The executable program rosenbrock reads in the `params.in.X` file and evaluates the objective function at the given values for x_1 and x_2. Then, rosenbrock writes out the objective function data to the `results.out.X` file. Here is the listing for the file `results.out.1`:

```
    2.4200000000e+01 f
```

The value shown above is the value of the objective function, and the descriptor 'f' is an optional tag returned by the simulation code. When the system call has completed, DAKOTA reads in the data from the `results.in.X` file. Then, DAKOTA continues with executions of the rosenbrock program until the optimization process is complete.

2.4.11 Optimization with a User-Supplied Simulation Code - Case 2

In many situations the user-supplied simulation code cannot be modified to read and write the `params.in.X file` and the `results.out.X` file, as described above. Typically, this occurs when the simulation code is a commercial or proprietary software product that has specific input file and output file formats. In such cases, it is common to replace the executable program name in the DAKOTA input file with the name of a UNIX shell script containing a sequence of commands that read and write the necessary files and run the simulation code. For example, the executable program named `rosenbrock` listed in Figure 2.28 could be replaced by a UNIX C-shell script named `simulator_script`, with the script containing a sequence of commands to perform the following steps: insert the data from the `parameters.in.X` file into the input file of the simulation code, execute the simulation code, post process the files generated by the simulation code to compute response data, and return the response data to DAKOTA in the `results.out.X` file. The steps that are typically used in constructing and using a UNIX shell script are described in Section 16.1.

2.5 Where to Go from Here

This chapter has provided an introduction to the basic capabilities of DAKOTA including parameter studies, various types of optimization, and uncertainty quantification sampling. More information on the DAKOTA input file syntax is provided in the remaining chapters in this text and in the DAKOTA Reference Manual [17]. Additional example problems that demonstrate some of DAKOTA's advanced capabilities are provided in Chapter 10, Chapter 13, Chapter16, and Chapter 20.

Here are a few pointers to sections of this manual that many new users find useful:

- Chapter 7 describes the different DAKOTA output file formats, including commonly encountered error messages.

- Chapter 16 demonstrates how to employ DAKOTA with a user-supplied simulation code.
  Most DAKOTA users will follow the approach described in this chapter.
- Chapter 17 provides guidelines on how to choose an appropriate optimization, uncertainty quantification, or parameter study method based on the characteristics of your application.
- Chapter 18 describes the file restart and data re-use capabilities of DAKOTA.

Chapter 3

DAKOTA Capabilities

3.1 Overview

This chapter provides a brief, but comprehensive, overview of DAKOTA's capabilities. Additional details and example problems are provided in subsequent chapters in this manual.

3.2 Parameter Study Methods

Parameter studies are often performed to explore the effect of parametric changes within simulation models. DAKOTA provides four parameter study methods that may be selected by the user.

Multidimensional: Forms a regular lattice or grid in an n-dimensional parameter space, where the user specifies the number of intervals used for each parameter.

Vector: Performs a parameter study along a line between any two points in an n-dimensional parameter space, where the user specifies the number of steps used in the study.

Centered: Given a point in an n-dimensional parameter space, this method evaluates nearby points along the coordinate axes of the parameter space. The user selects the number of steps and the step size.

List: The user supplies a list of points in an n-dimensional space where DAKOTA will evaluate response data from the simulation code.

Additional information on these methods is provided in Chapter 8.

3.3 Sampling Methods and Design of Experiments

Sampling methods and design of experiments are often used to explore the parameter space of an engineering design problem. Two software packages are available in DAKOTA for performing these studies, LHS and DDACE, both of which were developed at Sandia Labs.

LHS (Latin Hypercube Sampling): This package provides both Monte Carlo (random) sampling and latin hypercube sampling methods, which can be used with probabilistic variables in DAKOTA that have the following distributions: Gaussian (normal), lognormal, uniform, loguniform, Weibull, and user-supplied histograms. In addition, the user can supply a correlation matrix for the variables to account for correlations among the variables [45]. The LHS package currently serves two purposes: (1) it can be used for uncertainty quantification by sampling over uncertain variables characterized by probability distributions (see Section 3.4), or (2) it can be used in a DACE mode in which any design and state variables are treated as having uniform distributions (see the `all_variables` flag in the Reference Manual [17]). The LHS package comes in two versions: "old" (circa 1980) and "new" (circa 1998), where only the former may currently be distributed externally.

DDACE (Distributed Design and Analysis of Computer Experiments): The DACE package includes both stochastic sampling methods and classical design of experiments methods [64]. The stochastic methods are Monte Carlo (random) sampling, latin hypercube sampling, and orthogonal array sampling. The DDACE package currently supports variables that have either normal or uniform distributions. However, only the uniform distribution is available in the DAKOTA interface to DDACE. The classical design of experiments methods in DDACE are central composite design (CCD) and Box-Behnken (BB) sampling. A grid-based sampling method also is available. DDACE is available under a GNU Lesser General Public License and is distributed with DAKOTA.

Additional information on these methods is provided in Chapter 9.

3.4 Uncertainty Quantification

Uncertainty quantification methods (also referred to as nondeterministic analysis methods) involve the computation of probabilistic information about response functions based on sets of simulations taken from the specified probability distributions for uncertain input parameters. Put another way, these methods perform a forward uncertainty propagation in which probability information for input parameters is mapped to probability information for output response functions. The UQ methods in DAKOTA include various sampling-based approaches (e.g., Monte Carlo and Latin hypercube sampling) discussed previously in Section 3.3, along with analytic reliability methods and stochastic finite element methods.

Analytic Reliability Methods: This suite of methods includes the Advanced Mean Value Method (AMV), the iterated Advanced Mean Value Method (AMV+), and the First Order Reliability Method (FORM). For the AMV and AMV+ methods, the user now has the option to specify either probability levels or reliability levels in the DAKOTA input file. Efforts are currently underway to implement the Second Order Reliability Method (SORM). In versions 3.0 and 3.1 of DAKOTA, the AMV and AMV+ methods were dependent on the NPSOL optimization software package. This dependence has been removed, and the user now has a choice between NPSOL and OPT++.

Stochastic Finite Element Methods: The objective of these techniques is to characterize the response of systems whose governing equations involve stochastic coefficients. The development of these techniques mirrors that of deterministic finite element analysis utilizing the notions of projection, orthogonality, and weak convergence [29], [30].

Additional information on these methods is provided in Chapter 10.

3.5 Optimization Software Packages

Several optimization software packages have been integrated with DAKOTA. These include freely-available software packages developed by research groups external to Sandia Labs, Sandia-developed software that has been released to the public under GNU licenses, and commercially-developed software. These optimization software packages provide the DAKOTA user with access to well-tested, proven methods for use in engineering design applications, as well as access to some of the newest developments in optimization algorithm research.

COLINY: Methods for nongradient-based local and global optimization which utilize the Common Optimization Library INterface (COLIN). This algorithm library supersedes the SGOPT library. COLINY currently includes evolutionary algorithms (including several genetic algorithms and Evolutionary Pattern Search), simple pattern search, Monte Carlo sampling, and the DIRECT and Solis-Wets algorithms. COLINY also include interfaces to third-party optimizers APPS [44] and COBYLA2. This software is available to the public under a GNU Lesser General Public License (LGPL) through ACRO (A Common Repository for Optimizers) and the source code for COLINY is included with DAKOTA (web page: http://www.cs.sandia.gov/Acro).

CONMIN (CONstrained MINimization): Methods for gradient-based constrained and unconstrained optimization [65]. The constrained optimization algorithm is the method of feasible directions (MFD) and the unconstrained optimization algorithm is the Fletcher-Reeves conjugate gradient (CG) method. This software is freely available to the public from NASA, and the CONMIN source code is included with DAKOTA.

DOT (Design Optimization Tools): Methods for gradient-based optimization for constrained and unconstrained optimization problems [67]. The algorithms available for constrained optimization are modified-MFD, SQP, and sequential linear programming (SLP). The algorithms available for unconstrained optimization are the Fletcher-Reeves CG method and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) quasi-Newton technique. DOT is a commercial software product of Vanderplaats Research and Development, Inc. (web page: http://www.vrand.com). Sandia National Laboratories and Los Alamos National Laboratory have limited seats for DOT. *Other users may obtain their own copy of DOT and compile it with the DAKOTA source code by following the steps given in the file /Dakota/INSTALL.*

JEGA: SOGA/MOGA (Single- or Multi-Objective Genetic Algorithm): John Eddy (member of technical staff at Sandia) implemented both single- and multi-objective optimization methods that employ genetic algorithms. The SOGA method provides a basic GA optimization capability that uses many of the same software elements as the MOGA method. See details on MOGA below.

MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization): formerly known as rSQP++, MOOCHO provides both general-purpose gradient-based algorithms for nested analysis and design (NAND) and large-scale gradient-based optimization algorithms for simultaneous analysis and design (SAND). This software is not yet available to the public.

NPSOL: Methods for gradient-based constrained and unconstrained optimization problems using a sequential quadratic programming (SQP) algorithm [31]. NPSOL is a commercial software product of Stanford University (web site: www.sbsi-sol-optimize.com). Sandia National Laboratories, Lawrence Livermore National Laboratory, and Los Alamos National Laboratory all have site licenses for NPSOL. *Other users may obtain their own copy of NPSOL and compile it with the DAKOTA source code by following the steps given in the file /Dakota/INSTALL.*

OPT++: Methods for gradient-based and nongradient-based optimization of unconstrained, bound-constrained, and nonlinearly constrained optimization problems [50]. OPT++ includes a variety of Newton-based methods (quasi-Newton, finite-difference Newton, Gauss-Newton, and full-Newton), as well as the Polak-Ribeire CG method and the parallel direct search (PDS) method. OPT++ now contains a nonlinear interior point algorithm for handling general constraints. OPT++ is an active research tool and new optimization capabilities are continually being added to its suite of capabilities. OPT++ is available to the public under the GNU LGPL and the source code is included with DAKOTA (web page: http://csmr.ca.sandia.gov/projects/opt++/opt++.html).

PICO (Parallel Integer Combinatorial Optimization): PICO's branch-and-bound algorithm is available in DAKOTA for use on nonlinear optimization problems involving discrete variables or a combination of continuous and discrete variables [16]. PICO is available to the public under the GNU LGPL and the source code is included with DAKOTA (web page: http://www.cs.sandia.gov/PICO). Note: PICO's methods for linear programming are not available under DAKOTA.

SGOPT (Stochastic Global OPTimization): Access to this library within DAKOTA has been deprecated; the methods have been migrated to the COLINY library.

Additional information on these methods is provided in Chapter 11.

3.6 Additional Optimization Capabilities

The optimization software packages described above provide algorithms to handle a wide variety of optimization problems. This includes algorithms for constrained and unconstrained optimization, as well as algorithms for gradient-based and nongradient-based optimization. Listed below are additional optimization capabilities that are available in DAKOTA.

MOGA - Multiobjective Optimization with Genetic Algorithms (without Weight Factors): The MOGA package allows for the formulation of multiobjective optimization problems without the user specifying weights on the various objective function values. The MOGA method identifies non-dominated design points that lie on the Pareto front using a genetic algorithm search method. The advantage of the MOGA method versus conventional multiobjective optimization with weight factors (see below), is that MOGA finds points along the entire Pareto front whereas the multiobjective optimization method produces only a single point on the Pareto front. The advantage of the MOGA method versus the Pareto-set optimization strategy is that MOGA is better able to find points on the Pareto front when the Pareto front is non convex. However, the use of a GA search method in MOGA causes the MOGA method to be much more computationally expensive than conventional multiobjective optimization using weight factors.

Multiobjective Optimization (with Weight Factors): In multiobjective optimization, a composite objective function is constructed from a set of individual objective functions. The user can specify the scalar weight factors that are applied to the individual objective functions in computing the composite objective function. This approach works with any of the optimization methods listed in Section 3.5. Also, both constrained and unconstrained multiobjective optimization problems can be formulated and solved with DAKOTA. Note that multiobjective optimization is related to the Pareto-set optimization strategy described in Section 3.8, with the difference that the former computes a single optimum and the latter computes a set of optima in order to generate a Pareto trade-off surface.

Simultaneous Analysis and Design (SAND): In SAND, one converges the optimization process at the same time as converging a nonlinear simulation code. In this approach, the solution of the simulation code (often a system of ordinary or partial differential equations) is posed as a set of equality constraints in the optimization problem and these equality constraints are only satisfied by the optimizer in the limit. This formulation necessitates a close coupling between DAKOTA and the simulation code so that the internal vectors and matrices from the simulation code (in particular, the residual vector and its state and design Jacobian matrices) are available to the SAND optimizer. This approach has the potential to reduce the cost of optimization significantly since the nonlinear simulation is only converged once, instead of on every function evaluation. The drawback is that this approach requires substantial software modifications to the simulation code; something that can be impractical in some cases and impossible in others. A new SAND capability employing the MOOCHO library is under development that will intrusively couple DAKOTA with multiphysics simulation frameworks under development at Sandia.

Additional information on these methods is provided in Chapter 11.

3.7 Nonlinear Least Squares for Parameter Estimation

Nonlinear least squares methods are optimization algorithms which exploit the special structure of a least squares objective function (see Section 1.4.2). These problems commonly arise in parameter estimation and test/analysis reconciliation. In practice, least squares solvers will tend to converge more rapidly than general-purpose optimization algorithms when the residual terms in the least squares formulation tend towards zero at the solution. Least squares solvers may experience difficulty when the residuals at the solution are significant, although experience has shown that the NL2SOL method can handle some problems that are highly nonlinear and have nonzero residuals at the solution.

[. . . add Dennis/Gay/Welsch NL2SOL ref to bibliography – could not save bibliography changes on 20-May]

NL2SOL: The NL2SOL algorithm **[REF]** uses a secant-based algorithm to solve least-squares problems. In practice, it is more robust to nonlinear functions and nonzero residuals than conventional Gauss-Newton algorithms.

Gauss-Newton: DAKOTA's Gauss-Newton algorithm utilizes the Hessian approximation described in Section 1.4.2. The exact objective function value, exact objective function gradient, and the approximate objective function Hessian are defined from the least squares term values and gradients and are passed to the full-Newton optimizer from the OPT++ software package. As for all of the Newton-based optimization algorithms in OPT++, unconstrained, bound-constrained, and generally-constrained problems are supported. However, for the generally-constrained case, a derivative order mismatch exists in that the nonlinear interior point full Newton algorithm will require second-order information for the nonlinear constraints whereas the Gauss-Newton approximation only requires first order information for the least squares terms.

NLSSOL: The NLSSOL algorithm is a commercial software product of Stanford University (web site: http://www.sbsi-sol-optimize.com) that is bundled with current versions of the NPSOL library. It uses an SQP-based approach to solve generally-constrained nonlinear least squares problems. It periodically employs the Gauss-Newton Hessian approximation to accelerate the search. It requires only first-order information for the least squares terms and nonlinear constraints. Sandia National Laboratories, Lawrence Livermore National Laboratory, and Los Alamos National Laboratory all have site licenses for NLSSOL. *Other users may obtain their own copy of NLSSOL and compile it with the DAKOTA source code by following the NPSOL installation steps given in the file /Dakota/INSTALL.*

Additional information on these methods is provided in Chapter 12.

3.8 Optimization Strategies

Due to the flexibility of DAKOTA's object-oriented design, it is relatively easy to create algorithms that combine several of DAKOTA's capabilities. These algorithms are referred to as *strategies*:

Multilevel Hybrid Optimization: This strategy allows the user to specify a sequence of optimization methods, with the results from one method providing the starting point for the next method in the sequence. An example which is useful in many engineering design problems involves the use of a nongradient-based global optimization method (e.g., genetic algorithm) to identify a promising region of the parameter space, which feeds its results into a gradient-based method (quasi-Newton, SQP, etc.) to perform an efficient local search for the optimum point.

Multistart Local Optimization: This strategy uses many local optimization runs (often gradient-based), each of which is started from a different initial point in the parameter space. This is an attractive strategy in situations where multiple local optima are known to exist or may potentially exist in the parameter space. This approach combines the efficiency of local optimization methods with the parameter space coverage of a global stratification technique.

Pareto-Set Optimization: The Pareto-set optimization strategy allows the user to specify different sets of weights for the individual objective functions in a multiobjective optimization problem. DAKOTA executes each of these weighting sets as a separate optimization problem, serially or in parallel, and then outputs the set of optimal designs which define the Pareto set. Pareto set information can be useful in making trade-off decisions in engineering design problems.

[Note that the MOGA algorithm (see above) also provides a means to identify points on the Pareto front.]

Mixed Integer Nonlinear Programming (MINLP): This strategy uses the branch and bound capabilities of the PICO package to perform optimization on problems that have both discrete and continuous design variables. PICO provides a branch and bound engine targeted at mixed integer linear programs (MILP), which when combined with DAKOTA's nonlinear optimization methods, results in a MINLP capability. In addition, the multiple NLPs solved within MINLP provide an opportunity for concurrent execution of multiple optimizations.

Surrogate-Based Optimization (SBO): This strategy combines the sampling methods, approximation methods, and optimization capabilities of DAKOTA. The SBO strategy is particularly effective on real-world engineering design problems that contain nonsmooth features (e.g., slope discontinuities, multiple local minima) where gradient-based optimization methods often have trouble. In SBO, the optimization algorithm operates on a surrogate model instead of directly operating on the computationally expensive simulation model. The surrogate model can be formed from data samples and surface fitting methods (see Section 3.9), or it can be a simplified (e.g., coarsened finite element mesh, less detailed) version of the original computational model. For either type of surrogate model, the SBO algorithm periodically checks the accuracy of the surrogate model against the original high-fidelity model. The SBO strategy in DAKOTA can be implemented using heuristic rules (less expensive) or a strategy that is guaranteed to converge (more expensive). The development of SBO strategies is an area of active research in the DAKOTA project.

Optimization Under Uncertainty (OUU): Many real-world engineering design problems contain stochastic features and must be treated using OUU methods such as robust design and reliability-based design. For OUU, the uncertainty quantification methods of DAKOTA are combined with optimization algorithms. This allows the user to formulate problems where one or more of the objective and constraints are stochastic. Due to the computational expense of both optimization and UQ, the simple nesting of these methods in OUU can be computationally prohibitive for real-world design problems. For this reason, surrogate-based OUU methods have been developed which can reduce the overall expense by an order of magnitude or more. OUU methods are an active research area.

These strategies are covered in more detail in Chapter 13.

3.9 Surface Fitting Methods

Surface fitting methods, often referred to as *response surface methods*, can be used to explore the variations in response quantities over regions of the parameter space. In addition, the surfaces can serve as surrogate models for optimization studies (see the surrogate-based optimization strategy in Section 3.8). The surface fitting methods in DAKOTA include software developed by Sandia researchers and by various researchers in the academic community. These surface fitting methods work in conjunction with the sampling methods and design of experiments methods described in Section 3.3.

Taylor Series Expansion: This is a local first-order or second-order model centered at a point in the parameter space.

Polynomial Regression: First-order (linear), second-order (quadratic), and third-order (cubic) polynomial response surfaces computed using linear least squares regression methods. Note: there is currently no use of forward- or backward-stepping regression methods to eliminate unnecessary terms from the polynomial model.

Kriging Interpolation: An implementation of spatial interpolation using kriging methods and Gaussian correlation functions [36]. The algorithm used in the kriging process generates a C^2-continuous surface that exactly interpolates the data values.

Artificial Neural Networks: An implementation of the stochastic layered perceptron neural network developed by Prof. D. C. Zimmerman of the University of Houston [72]. This neural network method is intended to have a lower training (fitting) cost than typical neural networks.

Multivariate Adaptive Regression Splines (MARS): Software developed by Prof. J. H. Friedman of Stanford University [27]. The MARS method creates a C^2-continuous patchwork of splines in the parameter space.

Additional information on these methods is provided in Chapter 14.

3.10 Parallel Computing

The methods and strategies in DAKOTA are designed to exploit parallel computing resources such as those found in a desktop multiprocessor workstation, a network of workstations, or a massively parallel computing platform. This parallel computing capability is a critical technology for rendering real-world engineering design problems computationally tractable. DAKOTA employs the concept of *multilevel parallelism*, which takes simultaneous advantage of opportunities for parallel execution from multiple sources:

Parallel Simulation Codes: DAKOTA works equally well with both serial and parallel simulation codes.

Concurrent Execution of Analyses within a Function Evaluation: Some engineering design applications call for the use of multiple simulation code executions (different disciplinary codes, the same code for different load cases or environments, etc.) in order to evaluate a single response data set for a single set of parameters. If these simulation code executions are independent (or if coupling is enforced at a higher level), DAKOTA can perform them in parallel.

Concurrent Execution of Function Evaluations within an Iterator: With very few exceptions, the iterative algorithms described in Section 3.2 through Section 3.7 all provide opportunities for the concurrent evaluation of response data sets for different parameter sets. Whenever there exists a set of design point evaluations that are independent, DAKOTA can perform them in parallel.

Concurrent Execution of Iterators within a Strategy: Some of the DAKOTA strategies described in Section 3.8 generate a sequence of iterator subproblems. For example, the MINLP, Pareto-set, and multi-start strategies generate sets of optimization subproblems, and the optimization under uncertainty strategy generates sets of uncertainty quantification subproblems. Whenever these subproblems are independent, DAKOTA can perform them in parallel.

It is important to recognize that these four parallelism levels are nested, in that a strategy can schedule and manage concurrent iterators, each of which may manage concurrent function evaluations, each of which may manage concurrent analyses, each of which may execute on multiple processors. Additional information on parallel computing with DAKOTA is provided in Chapter 15.

3.11 Summary

DAKOTA is both a production tool for engineering design and analysis activities and a research tool for the development of new algorithms in optimization, uncertainty quantification, and related areas. Because of the extensible, object-oriented design of DAKOTA, it is relatively easy to add new iterative algorithms, strategies, simulation interfacing approaches, surface fitting methods, etc. In addition, DAKOTA can serve as a rapid prototyping tool for algorithm development. That is, by having a broad range of building blocks available (i.e., parallel computing, surrogate models, simulation interfaces, fundamental algorithms, etc.), new capabilities can be assembled rapidly which leverage the previous software investments. For additional discussion on framework extensibility, refer to the DAKOTA Developers Manual [18].

The capabilities of DAKOTA have been used to solve engineering design and optimization problems at Sandia Labs, at other Department of Energy labs, and by our industrial and academic collaborators. Often, this real-world experience has provided motivation for research into new areas of optimization. The DAKOTA development team welcomes feedback on the capabilities of this software toolkit, as well as suggestions for new areas of research.

Chapter 4

Variables

4.1 Overview

The variables section in a DAKOTA input file specifies the parameter set to be iterated by a particular method. In the case of an optimization study, these variables are adjusted in order to locate an optimal design; in the case of parameter studies/sensitivity analysis/design of experiments, these parameters are perturbed to explore the parameter space; and in the case of uncertainty analysis, the variables are associated with probabilistic characterizations which are used to quantify the uncertainty in response functions. To accommodate these and other types of studies, DAKOTA supports design, uncertain, and state variable types for continuous and discrete variable domains.

This chapter will present a brief overview of the types of variables and their uses, as well as cover some user issues relating to integer/discrete conversions, file formats, and the active set vector. For a detailed description of variables section syntax and example specifications, refer to the variables commands chapter in the DAKOTA Reference Manual [17].

4.2 Design Variables

Design variables are those variables which are modified for the purposes of computing an optimal design. These variables may be continuous (real-valued) or discrete (integer-valued).

4.2.1 Continuous Design Variables

The most common type of design variables encountered in engineering applications are of the continuous type. These variables may assume any real value (e.g., `12.34`, `-1.735e+07`) within their bounds. All but a handful of the optimization algorithms in DAKOTA support continuous design variables exclusively.

4.2.2 Discrete Design Variables

Engineering design problems may contain discrete variables such as material types, feature counts, stock gauge selections, etc. These variables may assume only a fixed number of values within their bounds. While the general discrete variable case would allow this fixed set of values to include real numbers (e.g., x_1 can only assume the values `4.2`, `6.4`, and `8.5`), DAKOTA assumes that the discrete variables can be specified as a sequence of integers (e.g., x_1 can be `1`, `2`, or `3`) and that a mapping from the integer sequence to the discrete values can be applied if necessary within the user's interface. A common mapping is to use the integer value from DAKOTA as the index into a vector of discrete real values.

Discrete variables may be classified as either "noncategorical" or "categorical" discrete variables. In the former noncategorical case, the integrality condition can be relaxed during the solution process since the model can still compute meaningful response functions for non-integer values. For example, a discrete variable representing the thickness of a structure is generally a noncategorical variable since it can assume a continuous range of values during the algorithm iterations, even if it is desired to have a stock gauge thickness in the end. In the latter categorical case, the integrality cannot be relaxed since the model cannot obtain a solution for a non-integer value. For example, feature counts are generally categorical variables, since most computational models will not support a non-integer value for the number of instances of some feature (e.g., number of support brackets).

Gradient-based optimization methods cannot be directly applied to problems with discrete variables. For problems with noncategorical variables, branch and bound techniques can be used to relax the integrality conditions and apply gradient-based methods to a series of generated subproblems. For problems with categorical variables, nongradient-based methods (e.g., `coliny_ea`) are commonly used. Branch and bound techniques are discussed in Section 13.5 and nongradient-based methods are further described in Chapter 11.

In addition to engineering applications, many non-engineering applications in the fields of scheduling, logistics, and resource allocation contain discrete design parameters. Within the Department of Energy, solution techniques for these problems impact programs in stockpile evaluation and management, production planning, nonproliferation, transportation (routing, packing, logistics), infrastructure analysis and design, energy production, environmental remediation, and tools for massively parallel computing such as domain decomposition and meshing.

4.3 Uncertain Variables

Deterministic variables (i.e., those with a single known value) do not capture the behavior of the input variables in all situations. In many cases, the exact value of a model parameter is not precisely known. An example of such an input variable is the thickness of a heat treatment coating on a structural steel I-beam used in building construction. Due to variabilities and tolerances in the coating process, the thickness of the layer is known to follow a normal distribution with a certain mean and standard deviation as determined from experimental data. The inclusion of the uncertainty in the coating thickness is essential to accurately represent the resulting uncertainty in the response of the building.

Currently, uncertain variables in DAKOTA are modeled as continuous random variables, or in the case of histogram, with an empirical histogram representation. If a problem contains discrete random variables, then these variables can be modeled using the point-based histogram representation. The following types of uncertain variables are available:

- Normal: characterized by a mean and standard deviation. Also referred to as Gaussian. Bounded normal is also supported with an additional specification of lower and upper bounds.
- Lognormal: characterized by a mean and either a standard deviation or an error factor. The natural logarithm of a lognormal variable has a normal distribution. Bounded lognormal is also supported with an additional specification of lower and upper bounds.
- Uniform: characterized by a lower bound and an upper bound. Probability is constant between the bounds.
- Loguniform: characterized by a lower bound and an upper bound. The natural logarithm of a loguniform variable has a uniform distribution.
- Weibull: characterized by an alpha parameter and a beta parameter.

- Histogram: characterized by a set of (x, y) pairs that either map out histogram bins (a continuous interval with associated bin count) or histogram points (a discrete point value with associated count).

DAKOTA also supports a user-supplied correlation matrix to provide correlations among the uncertain input variables. By default, the correlation matrix is set to the identity matrix, i.e., no correlation among the uncertain variables.

For additional information on random variable probability distributions, refer to [41] and [71]. Refer to the DAKOTA Reference Manual [17] for more detail on the uncertain variable specifications and to Chapter 10 for a description of methods available to quantify the uncertainty in the response.

4.4 State Variables

State variables consist of "other" variables which are to be mapped through the simulation interface, in that they are not to be used for design and they are not modeled as being uncertain. State variables provide a convenient mechanism for parameterizing additional model inputs which, in the case of a numerical simulator, might include solver convergence tolerances, time step controls, or mesh fidelity parameters. Similar to the design variables discussed in Section 4.2, state variables can be continuous (real-valued) or discrete (integer-valued). For discrete variables which are not a sequence of integers, a mapping can be applied between the integer and discrete values in the user's interface.

State variables, as with other types of variables, are viewed differently depending on the method in use. Since these variables are neither design nor uncertain variables, algorithms for optimization, least squares, and uncertainty quantification do not iterate on these variables; i.e., they are not active and are hidden from the algorithm. However, DAKOTA still maps these variables through the user's interface where they affect the computational model in use. This allows optimization, least squares, and uncertainty quantification studies to be executed under different simulation conditions (which will result, in general, in different results). Parameter studies and design of experiments methods, on the other hand, are general-purpose iterative techniques which do not draw a distinction between variable types. They include state variables in the set of variables to be iterated, which allows these studies to explore the effect of state variable values on the response data of interest.

In the future, state variables might be used in direct coordination with an optimization, least squares, or uncertainty quantification algorithm. For example, state variables could be used to enact model adaptivity through the use of a coarse mesh or loose solver tolerances in the initial stages of an optimization with continuous model refinement as the algorithm nears the optimal solution.

4.5 Mixed Variables

The iterative method selected for use in DAKOTA determines what subset, or view, of the variables data is active in the iteration. The general case of having a mixture of various different types of variables is supported within all of the DAKOTA methods even though certain methods will only modify certain types of variables (e.g., optimizers and least squares methods only modify design variables, and uncertainty quantification methods only utilize uncertain variables). This implies that variables which are not under the direct control of a particular iterator will be mapped through the interface unmodified for all evaluations of the iterator. This allows for a variety of parameterizations within the model in addition to those which are being used by a particular iterator, which can provide the convenience of consolidating the control over various modeling parameters in a single file (the DAKOTA input file). An important related point is that the variable set that is active with a particular iterator is the same variable set for which derivatives are computed (see Section 6.3).

4.6 DAKOTA Parameters File Data Format

Application interfaces which employ system calls and forks to create separate simulation processes must communicate with the simulation through the file system. This is accomplished through the reading and writing of parameters and results files. DAKOTA uses its own format for this data input/output. Depending on the user's interface specification, DAKOTA will write the parameters file in either standard or APREPRO format. The former option uses a simple "`value tag`" format, whereas the latter option uses a "`{ tag = value }`" format for compatibility with the APREPRO utility [61].

4.6.1 Parameters file format (standard)

Prior to invoking a simulation, DAKOTA creates a parameters file which contains the current parameter values and a set of function requests. The standard format for this parameters file is shown in Figure 4.1.

where "`<int>`" denotes an integer value, "`<double>`" denotes a double precision value, and "`...`" indicates omitted lines for brevity. The first line specifies the total number of variables (`n`) with its identifier string "`variables`" followed by the number of functions (`m`) with its identifier string "`functions`." These integers are useful for dynamic memory allocation within a simulator or filter program. The next `n` lines specify the current values and descriptors of all of the variables within the parameter set *in the following order*: continuous design, discrete design, normal uncertain, lognormal uncertain, uniform uncertain, loguniform uncertain, weibull uncertain, histogram uncertain (bin histograms followed by point histograms), continuous state, and discrete state variables. The lengths of these vectors add to a total of n (that is, $n_{cdv} + n_{ddv} + n_{nuv} + n_{lnuv} + n_{uuv} + n_{luuv} + n_{wuv} + n_{huv} + n_{csv} + n_{dsv} = n$). If any of the variable types are not present in the problem, then its block is omitted entirely from the parameters file. The tags are the variable descriptors specified in the user's DAKOTA input file, or if no descriptors have been specified, default descriptors are used. The next `m` lines specify the request vector for each of the `m` functions in the response data set. These integer codes indicate what data is required on the current function evaluation and are described further in Section 4.7.

4.6.2 Parameters file format (APREPRO)

For the APREPRO format option, the same data is present and the same ordering is used as in the standard format. The only difference is that values are associated with their tags within "`{ tag = value }`" constructs as shown in Figure 4.2. This allows direct usage of these parameters files by either the APREPRO or DPREPRO utility, which are file pre-processors that can significantly simplify model parameterization. *[Note: APREPRO is a Sandia-developed pre-processor that is not distributed with DAKOTA. DPREPRO is a Perl script that performs many of the same functions as APREPRO, and DPREPRO is distributed with DAKOTA.]* When a parameters file in APREPRO format is included within a template file (using an include directive), the APREPRO utility recognizes these constructs as variable definitions which can then be used to populate targets throughout the template file [61].

4.7 The Active Set Vector

The active set vector contains a set of integer codes, one per response function, which describe the data needed on a particular execution of an interface. Integer values of 0 through 7 denote a 3-bit binary representation of all possible combinations of value, gradient, and Hessian requests for a particular function, with the most significant bit denoting the Hessian, the middle bit denoting the gradient, and the least significant bit denoting the value. The specific translations are shown in Table 4.1.

The active set vector in DAKOTA gets its name from managing the active set, i.e., the set of functions that are active on a particular function evaluation. However, it also manages the type of data that is needed

```
    <double> <var_tag_cdv$_1$>
    <double> <var_tag_cdv$_2$>
    ...
    <double> <var_tag_cdv$_n$>
    <int> <var_tag_ddv$_1$>
    <int> <var_tag_ddv$_2$>
    ...
    <int> <var_tag_ddv$_n$>
    <double> <var_tag_nuv$_1$>
    <double> <var_tag_nuv$_2$>
    ...
    <double> <var_tag_nuv$_n$>
    <double> <var_tag_lnuv$_1$>
    <double> <var_tag_lnuv$_2$>
    ...
    <double> <var_tag_lnuv$_n$>
    <double> <var_tag_uuv$_1$>
    <double> <var_tag_uuv$_2$>
    ...
    <double> <var_tag_uuv$_n$>
    <double> <var_tag_luuv$_1$>
    <double> <var_tag_luuv$_2$>
    ...
    <double> <var_tag_luuv$_n$>
    <double> <var_tag_wuv$_1$>
    <double> <var_tag_wuv$_2$>
    ...
    <double> <var_tag_wuv$_n$>
    <double> <var_tag_huv$_1$>
    <double> <var_tag_huv$_2$>
    ...
    <double> <var_tag_huv$_n$>
    <double> <var_tag_csv$_1$>
    <double> <var_tag_csv$_2$>
    ...
    <double> <var_tag_csv$_n$>
    <int> <var_tag_dsv$_1$>
    <int> <var_tag_dsv$_2$>
    ...
    <int> <var_tag_dsv$_n$>
    <int> ASV_1
    <int> ASV_2
    ...
    <int> ASV_m
```

Figure 4.1: Parameters file data format - standard option.

```
{ DAKOTA_VARS = <int> }
{ DAKOTA_FNS = <int> }
{ <var_tag_cdv$_1$> = <double> }
{ <var_tag_cdv$_2$> = <double> }
...
{ <var_tag_cdv$_n$> = <double> }
{ <var_tag_ddv$_1$> = <int> }
{ <var_tag_ddv$_2$> = <int> }
...
{ <var_tag_ddv$_n$> = <int> }
{ <var_tag_nuv$_1$> = <double> }
{ <var_tag_nuv$_2$> = <double> }
...
{ <var_tag_nuv$_n$> = <double> }
{ <var_tag_lnuv$_1$> = <double> }
{ <var_tag_lnuv$_2$> = <double> }
...
{ <var_tag_lnuv$_n$> = <double> }
{ <var_tag_uuv$_1$> = <double> }
{ <var_tag_uuv$_2$> = <double> }
...
{ <var_tag_uuv$_n$> = <double> }
{ <var_tag_luuv$_1$> = <double> }
{ <var_tag_luuv$_2$> = <double> }
...
{ <var_tag_luuv$_n$> = <double> }
{ <var_tag_wuv$_1$> = <double> }
{ <var_tag_wuv$_2$> = <double> }
...
{ <var_tag_wuv$_n$> = <double> }
{ <var_tag_huv$_1$> = <double> }
{ <var_tag_huv$_2$> = <double> }
...
{ <var_tag_huv$_n$> = <double> }
{ <var_tag_csv$_1$> = <double> }
{ <var_tag_csv$_2$> = <double> }
...
{ <var_tag_csv$_n$> = <double> }
{ <var_tag_dsv$_1$> = <int> }
{ <var_tag_dsv$_2$> = <int> }
...
{ <var_tag_dsv$_n$> = <int> }
{ ASV_1 = <int> }
{ ASV_2 = <int> }
...
{ ASV_m = <int> }
```

Figure 4.2: Parameters file data format - APREPRO option.

Table 4.1: Active set vector integer codes.

Integer Code	Binary representation	Meaning
7	111	Get Hessian, gradient, and value
6	110	Get Hessian and gradient
5	101	Get Hessian and value
4	100	Get Hessian
3	011	Get gradient and value
2	010	Get gradient
1	001	Get value
0	000	No data required, function is inactive

for functions that are active, and in that sense, has an extended meaning beyond that typically used in the optimization literature.

4.7.1 Active set vector control

Active set vector control may be turned off to allow the user to simplify the supplied interface by removing the need to check the content of the active set vector on each evaluation. The Interface Commands chapter in the Reference Manual provides additional information on this option (`deactivate active_set_vector`). Of course, this option trades some efficiency for simplicity and is most appropriate for those cases in which only a relatively small penalty occurs when computing and returning more data than may be needed on a particular function evaluation.

Chapter 5

Interfaces

5.1 Overview

The interface section in a DAKOTA input file specifies how function evaluations will be performed. The mechanisms currently in place for performing function evaluations involve interfacing either with an application (i.e., a computational simulation code) or with an approximation (i.e., a surrogate-model).

In the case of a simulation code, the `application` interface is used to invoke the simulation with either system calls, forks, or direct function invocations. In the system call and fork cases, a separate process is created for the simulation and communication between DAKOTA and the simulation occurs through parameter and response files. For system call and fork interfaces, then, the interface section must also specify the details of this data transfer. In the direct function case, a separate process is not created and communication occurs directly through the function parameter list. Section 5.2 through Section 5.5 provide information on the application interfacing approaches.

In the case of use of an approximation in place of an expensive simulation code, an `approximation` interface can be selected to make use of surrogate modeling capabilities available within DAKOTA. Surrogate models are discussed further in Chapter 14.

This chapter will present an overview of the application interface procedures and components, as well as cover issues relating to file management and example data mappings. For a detailed description of interface section syntax, refer to the interface commands chapter in the DAKOTA Reference Manual [17].

5.2 The Direct Function Application Interface

The direct function interface capability may be used to invoke simulations which are linked into the DAKOTA executable. This interface eliminates overhead from process creation and file I/O and can simplify operations on massively parallel computers. These advantages are balanced with the practicality of converting an existing simulation code into a link library with a subroutine interface. Sandia's SALINAS structural dynamics code and Phoenix Integration's ModelCenter framework have been linked in this way, and a direct interface to Sandia's SIERRA multiphysics framework is under development. In the latter case, the additional effort is particularly justified since SIERRA unifies an entire suite of physics codes.

In addition to direct linking with simulation codes, the direct interface also provides access to internal polynomial test functions that are used for algorithm performance and regression testing. The following test functions are available: `textbook` (including `text_book1`, `text_book2`, `text_book3`, and `text_book_ouu`), `rosenbrock`, `cylinder_head`, and `cantilever`. While these functions are also available as external programs in the `/Dakota/test` directory, maintaining internally linked versions allows more rapid testing. See Chapter 20 for additional information on these test problems. An example input specification for a direct interface follows:

```
    interface,                                          \
            application direct,                         \
              analysis_driver = 'rosenbrock'
```

Additional specification examples are provided in Section 2.4, additional information on asynchronous usage of the direct function interface is provided in Section 15.3.1, and the details of adding a simulation code to the direct interface are provided in Section 16.2.

5.3 The System Call Application Interface

The system call approach invokes a simulation code or simulation driver by using the `system` function from the standard C library [46]. In this approach, the system call creates a new process which communicates with DAKOTA through parameter and response files. The system call approach allows the simulation to be initiated via its standard invocation procedure (as a "black box") and then coordinated with any variety of tools for pre- and post-processing. This approach has been widely used in previous studies [24], [25]. The system call approach involves more process creation and file I/O overhead than the direct function approach; however, this is most often of very little significance relative to the expense of the simulations. An example of a system call interface specification follows:

```
    interface,                                      \
            application system,                     \
              analysis_driver = 'text_book'         \
              parameters_file = 'text_book.in'      \
              results_file    = 'text_book.out'     \
              file_tag                              \
              file_save
```

More detailed examples of using the system call interface are provided in Section 2.4.10 and in Section 16.2, and information on asynchronous usage of the system call interface is provided in Section 15.3.2.

5.4 The Fork Application Interface

The fork application interface uses the `fork`, `exec`, and `wait` families of functions to manage simulation codes or simulation drivers. The `fork` or `vfork` calls create a copy of the DAKOTA process, `execvp` replaces this copy with the simulation code or driver process, and then DAKOTA uses the `wait` or `waitpid` functions to wait for completion of the new process. Transfer of variables and response data between DAKOTA and the simulator code or driver occurs through the file system in exactly the same manner as for the system call interface. An example of a fork interface specification follows:

```
    interface,                                           \
            application fork,                            \
              input_filter    = 'test_3pc_if'            \
              output_filter   = 'test_3pc_of'            \
              analysis_driver = 'test_3pc_ac'            \
              parameters_file = 'tb.in'                  \
              results_file    = 'tb.out'                 \
              file_tag
```

Information on asynchronous usage of the fork interface is provided in Section 15.3.3.

5.5 Fork or System Call: Which to Use?

The primary operational difference between the fork and system call application interfaces is that, in the fork interface, the `fork`/`exec` functions return a UNIX process identifier which can be utilized by the `wait`/`waitpid` functions to detect the completion of a simulation, whereas the system call application interface must use a response file detection scheme for this purpose. Thus, an important advantage of the fork interface over the system call interface is that it avoids the potential of a file race condition. This condition can occur when the responses file has been created but the writing of the response data set to this file has not been completed (see Section 15.3.2). While significant care has been taken to manage this file race condition in the system call case, the fork interface still has the potential to be more robust when performing function evaluations asynchronously.

Another advantage of the fork interface is that it has additional asynchronous capabilities when a function evaluation involves multiple analyses. As shown in Table 15.1, the fork interface supports asynchronous local and hybrid parallelism modes for managing concurrent analyses within function evaluations, whereas the system call interface does not. These additional capabilities again stem from the ability to track child processes by their UNIX process identifiers.

The only observed disadvantage to the fork interface in comparison to the system interface is that the `fork`/`exec`/`wait` functions are not part of the standard C library, whereas the `system` function is. As a result, support for implementations of the `fork`/`exec`/`wait` functions can vary from platform to platform. At one time, these commands were not available on some of Sandia's massively parallel computers. However, in the more mainstream UNIX environments, availability of `fork`/`exec`/`wait` should not be an issue.

In summary, the system call interface has been a workhorse for many years and is well tested and proven. However, the fork interface supports additional capabilities and is recommended when managing asynchronous simulation code executions. Having both interfaces available has proven to be useful on a number of occasions and they will both continue to be supported for the foreseeable future.

5.6 Interface Components

Figure 5.1 is an extension of Figure 1.1 which adds the detail of the components that make up each of the application interfaces (system call, fork, and direct). These components include an `input_filter` ("IFilter"), one or more `analysis_drivers`, and an `output_filter` ("OFilter"). The input and output filters provide optional facilities for managing simulation pre- and post-processing, respectively. More specifically, the input filter can be used to insert the DAKOTA parameters into the input files required by the simulator program, and the output filter can be used to recover the raw data from the simulation results and compute the desired response data set. If there is a single analysis code, it is often convenient to combine these pre- and post-processing functions into a single simulation driver script, and the separate input and output filter facilities are rarely used in this case. If there are multiple analysis drivers, however, the input and output filter facilities provide a convenient means for managing *nonrepeated* portions of the pre- and post-processing for multiple analyses. That is, pre- and post-processing tasks that must be performed for each analysis can be performed within the individual analysis drivers, and shared pre- and post-processing tasks that are only performed once for the set of analyses can be performed within the input and output filters.

When spawning function evaluations using system calls or forks, DAKOTA must communicate parameter and response data with the analysis drivers and filters through use of the file system. This is accomplished by passing the names of the parameters and results files on the command line when executing an analysis driver or filter. The input filter or analysis driver read data from the parameters file and the output filter or analysis driver write the appropriate data to the responses file. While not essential when the file names are fixed, the file names must be retrieved from the command line when DAKOTA is changing the file names from one function evaluation to the next (i.e., using UNIX temporary files or root names tagged

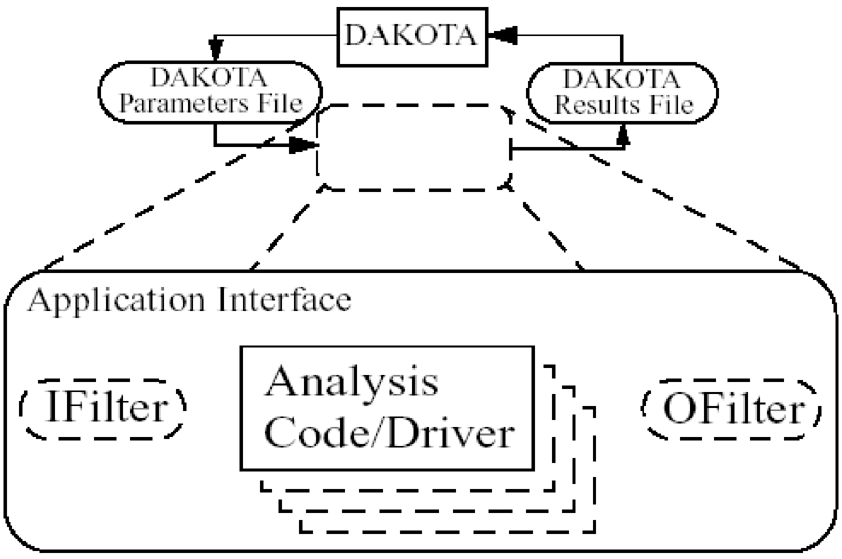


Figure 5.1: Components of the application interface

with numerical identifiers). In the case of a UNIX C-shell script, the two command line arguments are retrieved using `$argv[1]` and `$argv[2]` (see [1]). In the case of a C or C++ program, command line arguments are retrieved using `argc` (argument count) and `argv` (argument vector) [46], and for Fortran 77, the `iargc` function returns the argument count and the `getarg` subroutine returns command line arguments.

5.6.1 Single analysis driver without filters

If a single `analysis_driver` is selected in the interface specification to perform the complete parameters to responses mapping and filters are not needed (as indicated by omission of the `input_filter` and `output_filter` specifications), then only one process will appear in the execution syntax of the application interface. An example of this syntax in the system call case is:

```
(driver params.in results.out)
```

where "`driver`" is the user-specified analysis driver and "`params.in`" and "`results.out`" are the names of the parameters and results files, respectively, passed on the command line. In this case, the user need not retrieve the command line arguments since the same file names will be employed each time.

For the same mapping, the fork application interface echoes the following syntax:

```
blocking fork: driver params.in results.out
```

for which only a single blocking fork is needed to perform the evaluation.

Executing the same mapping with the direct application interface results in an echo of the following syntax:

```
Direct function: invoking driver
```

where this analysis driver must be linked as a function within DAKOTA's direct interface (see Section 16.2). Note that no files are involved for communication of parameter and response data, since this data is passed directly through the function parameter lists. Execution of the direct interface must currently be performed synchronously since multithreading is not yet supported.

Both the system call and fork interfaces support asynchronous operations. The asynchronous system call execution syntax involves executing the system call in the background:

```
(driver params.in.1 results.out.1) &
```

and the asynchronous fork execution syntax involves use of a nonblocking fork:

```
nonblocking fork: driver params.in.1 results.out.1
```

where file tagging (see Section 5.7.2) has been user-specified in both cases to prevent conflicts between concurrent analysis drivers. In these cases, the user must retrieve the command line arguments since the file names change on each evaluation.

5.6.2 Single analysis driver with filters

When filters are used, the syntax of the system call that DAKOTA performs is:

```
(ifilter params.in results.out;
     driver params.in results.out;
     ofilter params.in results.out)
```

in which the input filter ("`ifilter`"), analysis driver ("`driver`"), and output filter ("`ofilter`") processes are combined into a single system call through the use of semi-colons and parentheses (see [1]). All three portions are passed the names of the parameters and results files on the command line.

For the same mapping, the fork application interface echoes the following syntax:

```
blocking fork: ifilter params.in results.out;
     driver params.in results.out;
     ofilter params.in results.out
```

where a series of three blocking forks is used to perform the evaluation.

Executing the same mapping with the direct application interface results in an echo of the following syntax:

```
Direct function: invoking { ifilter driver ofilter }
```

where each of the three components must be linked as a function within DAKOTA's direct interface. Since asynchronous operations are not yet supported, execution simply involves invocation of each of the three linked functions in succession. Again, no files are involved since parameter and response data are passed directly through the function parameter lists.

Asynchronous executions would appear as follows for the system call interface:

```
(ifilter params.in.1 results.out.1;
     driver params.in.1 results.out.1;
     ofilter params.in.1 results.out.1) &
```

and, for the fork interface, as:

```
nonblocking fork: ifilter params.in.1 results.out.1;
     driver params.in.1 results.out.1;
     ofilter params.in.1 results.out.1
```

where file tagging of evaluations has again been user-specified in both cases. For the system call application interface, use of parentheses and semi-colons to bind the three processes into a single system call simplifies asynchronous process management compared to an approach using separate system calls. The fork application interface, on the other hand, does not rely on parentheses and accomplishes asynchronous operations by first forking an intermediate process. This intermediate process is then reforked for the execution of the input filter, analysis driver, and output filter. The intermediate process can be blocking or nonblocking (nonblocking in this case), and the second level of forks can be blocking or nonblocking (blocking in this case). The fact that forks can be reforked multiple times using either blocking or nonblocking approaches provides the enhanced flexibility to support a variety of parallelism models (see Chapter 15).

5.6.3 Multiple analysis drivers without filters

If a list of `analysis_drivers` is specified and filters are not needed (as indicated by omission of the `input_filter` and `output_filter` specifications), then the system call syntax would appear as:

```
(driver1 params.in results.out.1;
     driver2 params.in results.out.2;
     driver3 params.in results.out.3)
```

where "`driver1`", "`driver2`", and "`driver3`" are the user-specified analysis drivers and "`params.in`" and "`results.out`" are the user-selected names of the parameters and results files. Note that the results files for the different analysis drivers have been automatically tagged to prevent overwriting. This automatic tagging of *analyses* (see Section 5.7.4) is a separate operation from user-selected tagging of *evaluations* (see Section 5.7.2).

For the same mapping, the fork application interface echoes the following syntax:

```
blocking fork: driver1 params.in results.out.1;
     driver2 params.in results.out.2;
     driver3 params.in results.out.3
```

for which a series of three blocking forks is needed (no reforking of an intermediate process is required).

Executing the same mapping with the direct application interface results in an echo of the following syntax:

```
Direct function: invoking { driver1 driver2 driver3 }
```

where, again, each of these components must be linked within DAKOTA's direct interface and no files are involved for parameter and response data transfer.

Both the system call and fork interfaces support asynchronous function evaluations. The asynchronous system call execution syntax would be reported as

```
(driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3) &
```

and the nonblocking fork execution syntax would be reported as

```
nonblocking fork: driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3
```

where, in both cases, file tagging of evaluations has been user-specified to prevent conflicts between concurrent analysis drivers and file tagging of the results files for multiple analyses is automatically used. In the fork interface case, an intermediate process is forked to allow a non-blocking function evaluation, and this intermediate process is then reforked for the execution of each of the analysis drivers.

5.6.4 Multiple analysis drivers with filters

Finally, when combining filters with multiple `analysis_drivers`, the syntax of the system call that DAKOTA performs is:

```
(ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1)
```

in which all processes have again been combined into a single system call through the use of semi-colons and parentheses. Note that the secondary file tagging for the results files is only used for the analysis drivers and not for the filters. This is consistent with the filters' defined purpose of managing the non-repeated portions of analysis pre- and post-processing (e.g., overlay of response results from individual analyses; see Section 5.7.4 for additional information).

For the same mapping, the fork application interface echoes the following syntax:

```
blocking fork: ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1
```

for which a series of five blocking forks is used (no reforking of an intermediate process is required).

Executing the same mapping with the direct application interface results in an echo of the following syntax:

```
Direct function: invoking { ifilter driver1 driver2 driver3
     ofilter }
```

where each of these components must be linked as a function within DAKOTA's direct interface. Since asynchronous operations are not supported, execution simply involves invocation of each of the five linked functions in succession. Again, no files are involved for parameter and response data transfer since this data is passed directly through the function parameter lists.

Asynchronous executions would appear as follows for the system call interface:

```
(ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1) &
```

and for the fork interface:

```
nonblocking fork: ifilter params.in.1 results.out.1;
     driver1 params.in.1 results.out.1.1;
     driver2 params.in.1 results.out.1.2;
     driver3 params.in.1 results.out.1.3;
     ofilter params.in.1 results.out.1
```

where, again, user-selected file tagging of evaluations is combined with automatic file tagging of analyses. In the fork interface case, an intermediate process is forked to allow a non-blocking function evaluation, and this intermediate process is then reforked for the execution of the input filter, each of the analysis drivers, and the output filter.

5.7 File Management

This section describes some of the file management features that are employed during an execution of DAKOTA when file transfer of data is used for the communication between DAKOTA and the simulation code (i.e., when the system call or fork interfaces are used). These features can be used for generating unique filenames when utilizing DAKOTA's parallel execution capabilities and for debugging purposes when troubleshooting the interface between DAKOTA and the simulation code.

5.7.1 File Saving

The `file_save` option in the interface specification allows the user to control whether parameters and results files are retained or removed from the working directory. DAKOTA's default behavior is to remove files once their use is complete in order to not clutter the working directory. If the method output setting is verbose, a file remove notification will follow the function evaluation echo, e.g.:

```
(driver /usr/tmp/aaaa20305 /usr/tmp/baaa20305)
Removing /usr/tmp/aaaa20305 and /usr/tmp/baaa20305
```

However, by specifying `file_save` in the interface specification, these files will not be removed. This latter behavior is often useful for debugging communication between DAKOTA and simulator programs. An example of a `file_save` specification is shown in the file tagging example below.

5.7.2 File Tagging for Evaluations

When a user provides `parameters_file` and `results_file` specifications, the `file_tag` option in the interface specification allows the user to render the names of these parameters and results files unique by appending the function evaluation number to the root file names. Default behavior is to not tag these files, which has the advantage of allowing the user to ignore command line argument passing and always read to and write from the same file names. However, it has the disadvantage that files may be overwritten from one function evaluation to the next. By specifying `file_tag` in the interface specification, the file names become unique through the appended evaluation number. This uniqueness makes it necessary for the user's interface to retrieve the names of these files from the command line. The file tagging feature is most often used when concurrent simulations are running in a common disk space, since it can prevent conflicts between the simulations. An example specification of `file_tag` and `file_save` is shown below:

```
interface,                                        \
        application system,                       \
          analysis_driver =       'text_book'     \
          parameters_file =       'text_book.in'  \
          results_file    =       'text_book.out' \
          file_tag                                \
          file_save
```

Special case: When a user specifies names for the parameters and results files and `file_save` is used without `file_tag`, untagged files are used in the function evaluation but are then moved to tagged files after the function evaluation is complete in order to prevent overwriting files for which a `file_save` request has been given. If the output control is set to verbose, then a notification similar to the following will follow the function evaluation echo:

```
(driver params.in results.out)
Files with nonunique names will be tagged to enable
     file_save:
Moving params.in to params.in.1
Moving results.out to results.out.1
```

5.7.3 UNIX Temporary Files

If `parameters_file` and `results_file` are not specified by the user, then the default mechanisms for file communication are UNIX temporary files. For example, a system call to a single analysis driver would appear as:

```
    (driver /usr/tmp/aaaa20305 /usr/tmp/baaa20305)
```

and a system call to an analysis driver with filter programs would appear as:

```
    (ifilter /usr/tmp/aaaa22490 usr/tmp/baaa22490;
         driver /usr/tmp/aaaa22490 usr/tmp/baaa22490;
         ofilter /usr/tmp/aaaa22490 /usr/tmp/baaa22490)
```

These files have unique names as created by the `tmpnam` utility from the C standard library [46]. This uniqueness makes it a requirement for the user's interface to retrieve the names of these files from the command line. File tagging with evaluation number is unnecessary with UNIX temporary files (since they are already unique); thus, `file_tag` requests will be ignored. A `file_save` request will be honored, but it should be used with care since the temporary file directory could easily become cluttered without the user noticing.

5.7.4 File Tagging for Analysis Drivers

When multiple analysis drivers are involved in performing a function evaluation with either the system call or fork application interface, a secondary file tagging is *automatically* used in order to distinguish the results files used for the individual analyses. This applies to both the case of user-specified names for the parameters and results files and the default UNIX temporary file case. Examples for the former case were shown previously in Section 5.6.3 and Section 5.6.4. The following examples demonstrate the latter UNIX temporary file case. Even though Unix temporary files have unique names for a particular function evaluation, a tagging is still needed to manage the individual contributions of the different analysis drivers to the response results, since the same root results filename is used for each component. For the system call interface, the syntax would be similar to the following:

```
    (ifilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ;
         driver1 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.1;
         driver2 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.2;
         driver3 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.3;
         ofilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ)
```

and, for the fork interface, similar to:

```
    blocking fork:
         ifilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ;
         driver1 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.1;
         driver2 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.2;
         driver3 /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ.3;
         ofilter /var/tmp/aaawkaOKZ /var/tmp/baaxkaOKZ
```

The tagging of the results files with an analysis identifier is needed since each of the analysis drivers is responsible for contributing a user-defined subset of the total response results for the evaluation. If an output filter is not supplied, then DAKOTA will combine these portions through a simple overlaying of the individual contributions (i.e., summing the results in `/var/tmp/baaxkaOKZ.1`, `/var/tmp/baaxkaOKZ.2`, and `/var/tmp/baaxkaOKZ.3`). If this simple approach is inadequate, then an output filter should be supplied to perform the combination. This is the reason why the results file for the output filter does not use analysis tagging; it is responsible for the results combination (i.e., combining `/var/tmp/baaxkaOKZ.1`, `/var/tmp/baaxkaOKZ.2`, and `/var/tmp/baaxkaOKZ.3` into `/var/tmp/baaxkaOKZ`). In this case, DAKOTA will read only the results file from the output filter (i.e., `/var/tmp/baaxkaOKZ`) and interpret it as the total response set for the evaluation.

Parameters files are not currently tagged with an analysis identifier. This reflects the fact that DAKOTA does not attempt to subdivide the requests in the active set vector for different analysis portions. Rather, the total active set vector is passed to each analysis driver and the appropriate subdivision of work *must be defined by the user*. This allows the division of labor to be very flexible. In some cases, this division might occur across response functions, with different analysis drivers managing the data requests for different response functions. And in other cases, the subdivision might occur within response functions, with different analysis drivers contributing portions to each of the response functions. The only restriction is that each of the analysis drivers must follow the response format dictated by the total active set vector. For response data for which an analysis driver has no contribution, 0's must be used as placeholders.

5.8 Parameter to Response Mappings

Following are several examples of interface mappings as evidenced by the parameters files and corresponding results files. A typical input file for 2 variables ($n = 2$) and 3 functions ($m = 3$) using the standard parameters file format (see Section 4.6.1) is as follows:

```
2 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
1 ASV_1
1 ASV_2
1 ASV_3
```

where the numerical values are associated with their tags within "`value tag`" constructs. The number of design variables (n) and the string "`variables`" are followed by the number of functions (m) and the string "`functions`", the values of the design variables and their tags, and the active set vector (ASV) and its tags. The descriptive tags for the variables are always present and they are either the descriptors specified in the user's variables specification or are default descriptors if none were provided. The length of the active set vector is equal to the number of functions (m). In the case of an optimization data set with an objective function and two nonlinear constraints (three response functions total), the first ASV value is associated with the objective function and the remaining two are associated with the constraints (in whatever consistent constraint order has been defined by the user).

For the APREPRO format option (see Section 4.6.2), the same set of data appears as follows:

```
{ DAKOTA_VARS     = 2 }
{ DAKOTA_FNS      = 3 }
{ cdv_1           =  1.5000000000e+00 }
{ cdv_2           =  1.5000000000e+00 }
{ ASV_1           =                 1 }
{ ASV_2           =                 1 }
{ ASV_3           =                 1 }
```

where the numerical values are associated with their tags within "`{ tag = value }`" constructs.

The user-supplied application interface, comprised of a simulator program or driver and (optionally) filter programs, is responsible for reading the parameters file and creating a results file that contains the response data requested in the ASV. This response data is written in the format described in Section 6.2. Since the ASV contains all ones in this case, the response file corresponding to the above input file would contain values for the three functions:

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
```

Since function tags are optional, the following would be equally acceptable:

```
1.2500000000e-01
1.5000000000e+00
1.7500000000e+00
```

For the same parameters with different ASV components,

```
2 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
3 ASV_1
3 ASV_2
3 ASV_3
```

the following response data is required:

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
[ 5.0000000000e-01 5.0000000000e-01 ]
[ 3.0000000000e+00 -5.0000000000e-01 ]
[ 0.0000000000e+00 3.0000000000e+00 ]
```

Here, we need not only the function values, but also each of their gradients. Another modification to the ASV components yields the following parameters file,

```
2 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
2 ASV_1
0 ASV_2
2 ASV_3
```

for which the following results file is needed:

```
[ 5.0000000000e-01 5.0000000000e-01 ]
[ 0.0000000000e+00 3.0000000000e+00 ]
```

Here, we need gradients for functions `f` and `c2`, but not for `c1`, presumably since this constraint is inactive.

A full Newton optimizer might well make the following request:

```
2 variables 1 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
7 ASV_1
```

for which the following results file (containing the objective function, its gradient vector, and its Hessian matrix) is needed:

```
1.2500000000e-01 f
[ 5.0000000000e-01 5.0000000000e-01 ]
[[ 3.0000000000e+00 0.0000000000e+00 0.0000000000e+00
     3.0000000000e+00 ]]
```

Lastly, a more advanced example might have multiple types of variables present:

```
11 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
2 ddv_1
2 ddv_2
2 ddv_3
3.5000000000e+00 csv_1
3.5000000000e+00 csv_2
3.5000000000e+00 csv_3
3.5000000000e+00 csv_4
4 dsv_1
4 dsv_2
3 ASV_1
3 ASV_2
3 ASV_3
```

In this case, the required length of the gradient vectors depends upon the type of study being performed (see Section 6.3). In an optimization problem, gradients are only needed with respect to the continuous design variables, in which case the following response data would be appropriate ($n_{grad} = 2$):

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
[ 5.0000000000e-01 5.0000000000e-01 ]
[ 3.0000000000e+00 -5.0000000000e-01 ]
[ 0.0000000000e+00 3.0000000000e+00 ]
```

In a parameter study, however, no distinction is drawn between different types of continuous variables, and gradients would be needed with respect to all continuous variables ($n_{grad} = 6$), e.g.:

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
[ 5.0000000000e-01 5.0000000000e-01 6.2500000000e+01
     6.2500000000e+01 6.2500000000e+01 6.2500000000e+01 ]
[ 3.0000000000e+00 -5.0000000000e-01 0.0000000000e+00
     0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 ]
[ 0.0000000000e+00 3.0000000000e+00 0.0000000000e+00
     0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 ]
```

Chapter 6

Response Data

6.1 Overview

The `responses` specification in a DAKOTA input file specifies the types of data that can be returned from an interface during DAKOTA's execution. The specification includes the number and type of response functions (objective functions, nonlinear constraints, least squares terms, etc.) as well as availability of first and second derivatives (gradient vectors and Hessian matrices) for these response functions.

This chapter will present a brief overview of the response data sets and their uses, as well as cover some user issues relating to file formats and derivative vector and matrix sizing. For a detailed description of responses section syntax and example specifications, refer to the responses commands chapter in the DAKOTA Reference Manual [17].

6.1.1 Response function types

The types of response functions specified in the responses specification depend on the iterative technique specified in the method specification:

- `num_objective_functions`, `num_nonlinear_inequality_constraints`, `num_nonlinear_equality_constraints`: this is an optimization data set for use with optimization methods from DOT, NPSOL, CONMIN, OPT++, and COLINY.
- `num_least_squares_terms`, `num_nonlinear_inequality_constraints`, `num_nonlinear_equality_constraints`: this is a least squares data set for use with Gauss-Newton and NLSSOL.
- `num_response_functions`: this is a generic data set for use with uncertainty quantification methods.

Certain general-purpose iterative techniques, such as parameter studies and design of experiments methods, can be used with any of these data sets.

6.1.2 Gradient availability

Gradient availability for these response functions may be described by:

- `no_gradients`: gradient data is not needed.

- `numerical_gradients`: gradient data is needed and will be computed by finite differences.
- `analytic_gradients`: gradient data is needed and is available directly from the simulation code (finite differencing is not required).
- `mixed_gradients`: some gradient information is available directly from the simulation whereas the rest will have to be finite differenced.

The gradient specification also links back to the iterative method being employed. Gradient data is commonly needed when the iterative study involves gradient-based optimization, uncertainty quantification with analytic reliability methods, or local sensitivity analysis.

6.1.3 Hessian availability

Hessian availability for the response functions has a subset of the gradient availability specifications:

- `no_hessians`: Hessian data is not needed.
- `analytic_hessians`: Hessian data is needed and is available directly from the simulation code.

Numerical and mixed Hessians calculations are not currently supported. The Hessian specification also links back to the iterative method in use, and use of analytic Hessian data would commonly appear for gradient-based optimization using full Newton methods or, perhaps, for local sensitivity analysis.

6.2 DAKOTA Results File Data Format

Application interfaces which employ system calls and forks to create separate simulation processes must communicate with the simulation through the file system. This is accomplished through the reading and writing of parameters and results files. DAKOTA uses its own format for this data input/output. For the results file, only one format is supported (as compared to the two parameters file formats described in Section 4.6). Ordering of response functions is as listed in Section 6.1.1 (e.g., objective functions or least squares terms are first, followed by nonlinear inequality constraints, followed by nonlinear equality constraints).

After completion of a simulation, DAKOTA expects to read a file containing response data for the current set of parameters and corresponding to the current set of function requests in the active set vector. This response data must be in the following format:

The first block of data is the function values that have been requested, followed by a block of requested gradient data, followed by a block of requested Hessian data. Function data have no bracket delimiters and one character tag per function can be *optionally* supplied. These tags are not used by DAKOTA and are only included as an optional field for consistency with the parameters file format and for backwards compatibility. The tags are rendered optional through DAKOTA's use of regular expression pattern matching to detect whether an upcoming field is numerical data or a tag. If character tags are used, then they must be separated from data by either white space or new line characters and there must not be any white space embedded within a character tag (e.g., use "`variable1`" or "`variable_1`," but not "`variable 1`").

Function gradient vectors are delimited with single brackets [... n_{grad}-vector of doubles...]. Tags are not used and must not be present. White space separating the brackets from the data is optional.

Function Hessian matrices are delimited with double brackets [[... $n_{grad} \times n_{grad}$ matrix of doubles...]]. Data is listed by rows and can be run together or broken onto multiple lines for readability. Tags are not used and must not be present. White space separating the brackets from the data is optional, although white space must not appear between the double brackets.

```
<double> <fn_tag1>
<double> <fn_tag2>
...
<double> <fn_tagm>
[ <double> <double> .. <double> ]
[ <double> <double> .. <double> ]
...
[ <double> <double> .. <double> ]
[[ <double> <double> .. <double> ]]
[[ <double> <double> .. <double> ]]
...
[[ <double> <double> .. <double> ]]
```

Figure 6.1: Results file data format.

If the amount of data in the file does not match the function request vector, DAKOTA will abort with a response recovery format error message.

The format of the numeric fields may be floating point or scientific notation. In the latter case, acceptable exponent characters are "`E`" or "`e`." A common problem when dealing with Fortran programs is that a C++ read of a numeric field using "`D`" or "`d`" as the exponent (i.e., a double precision value from Fortran) may fail or be truncated. In this case, the "`D`" exponent characters must be replaced either through modifications to the Fortran source or compiler flags or through a separate post-processing step (e.g., using the UNIX `sed` utility).

6.3 Active Variables for Derivatives

An important question for proper management of both gradient and Hessian data is: if several different types of variables are used, **for which variables are response function derivatives needed?** That is, how is n_{grad} determined? The answer is that it depends on the iterative method in use. Methods determine what subset, or view, of the variables data is active in the iteration. The set of variables that is active in the iteration is the same set of variables for which derivatives are computed (see also Section 4.5).

Derivatives are never needed with respect to any discrete variables (since these derivatives do not exist) and the types of continuous variables for which derivatives are needed depend on the type of study being performed. For optimization and least squares problems, response function derivatives are only needed with respect to the *continuous design variables* ($n_{grad} = n_{cdv}$) since this is the information used by the optimizer in computing a search direction. Similarly, for nondeterministic analysis methods which use gradient and/or Hessian information, function derivatives are only needed with respect to the *uncertain variables* ($n_{grad} = n_{uv}$). And lastly, parameter study methods which are cataloguing gradient and/or Hessian information do not draw a distinction among continuous variables; therefore, function derivatives must be supplied with respect to *all continuous variables* that are specified ($n_{grad} = n_{cdv} + n_{uv} + n_{csv}$). This is generally not as complicated as it sounds, since it is common for optimization and least squares problems to only specify design variables and for nondeterministic analysis problems to only specify uncertain variables. DAKOTA allows for the specification of additional types of variables in these cases and DAKOTA will map these additional variables through the interface, but since they will not be used in the internal computations of the iterator, the derivatives of the function set with respect to the additional variables are not needed.

Chapter 7

Output from DAKOTA

7.1 Overview of Output Formats

Given an emphasis on complex numerical simulation codes that run on massively parallel supercomputers, DAKOTA's output has been designed to provide a succinct, text-based reporting of the progress of the iterations and function evaluations performed by an algorithm. In addition, DAKOTA provides a tabular output format that is useful for data visualization with external tools and a basic graphical output capability that is useful as a monitoring tool. Future work will include the development of a graphical user interface with more extensive capabilities.

7.2 Standard Output

DAKOTA outputs basic information to "standard out" (i.e., the screen) for each function evaluation, consisting of an evaluation number, parameter values, execution syntax, the active set vector, and the response data set. To describe the standard output of DAKOTA, optimization of the "container" problem (see Chapter 20 for problem formulation) is used as an example. The input file for this example is shown in Figure 7.1. In this example, there is one equality constraint, and DAKOTA's finite difference algorithm is used to provide central difference numerical gradients to the NPSOL optimizer.

```
# test file with a specific test.  The  is used to designate lines

strategy,                                                       \
        single_method                                           \
        tabular_graphics_data

method,                                                         \
        npsol_sqp

variables,                                                      \
        continuous_design = 2                                   \
          cdv_descriptor 'H' 'D'                                \
          cdv_initial_point 4.5 4.5                             \
          cdv_lower_bounds  0.0 0.0

interface,                                                      \
        system                                                  \
          analysis_driver = 'container'                         \
          parameters_file = 'container.in'                      \
          results_file    = 'container.out'                     \
          file_tag

responses,                                                      \
        num_objective_functions = 1                             \
        num_nonlinear_equality_constraints = 1                  \
        numerical_gradients                                     \
          method_source dakota                                  \
          interval_type central                                 \
          fd_gradient_step_size = 0.001                         \
        no_hessians
```

Figure 7.1: DAKOTA input file for the “container” example problem.

A partial listing of the output for the container optimization example follows:

```
Running MPI executable in serial mode.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
methodName = npsol_sqp
gradientType = numerical
Numerical gradients using central differences
to be calculated by the dakota finite difference routine.
hessianType = none

>>>>> Running Single Method Strategy.

>>>>> Running npsol_sqp iterator.




                     NPSOL  ---  Version 5.0-2      Sept 1995
                     ========================================

------------------------------------------
Begin Dakota derivative estimation routine
------------------------------------------

>>>>> Initial map for analytic portion of response:

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      4.5000000000e+00 H
                      4.5000000000e+00 D

(container container.in.1 container.out.1)

Active response data for function evaluation 1:
Active set vector = { 1 1 }
                      1.0713145108e+02 obj_fn
                      8.0444076396e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[1] + h:

------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                      4.5045000000e+00 H
                      4.5000000000e+00 D

(container container.in.2 container.out.2)

Active response data for function evaluation 2:
Active set vector = { 1 1 }
                      1.0719761302e+02 obj_fn
                      8.1159770472e+00 nln_eq_con_1
```

```
>>>>> Dakota finite difference gradient evaluation for x[1] - h:

------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                      4.4955000000e+00 H
                      4.5000000000e+00 D

(container container.in.3 container.out.3)

Active response data for function evaluation 3:
Active set vector = { 1 1 }
                      1.0706528914e+02 obj_fn
                      7.9728382320e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[2] + h:

------------------------------
Begin Function Evaluation    4
------------------------------
Parameters for function evaluation 4:
                      4.5000000000e+00 H
                      4.5045000000e+00 D

(container container.in.4 container.out.4)

Active response data for function evaluation 4:
Active set vector = { 1 1 }
                      1.0727959301e+02 obj_fn
                      8.1876180243e+00 nln_eq_con_1


>>>>> Dakota finite difference gradient evaluation for x[2] - h:

------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                      4.5000000000e+00 H
                      4.4955000000e+00 D

(container container.in.5 container.out.5)

Active response data for function evaluation 5:
Active set vector = { 1 1 }
                      1.0698339109e+02 obj_fn
                      7.9013403937e+00 nln_eq_con_1


>>>>> Total response returned to iterator:

Active set vector = { 3 3 }
                      1.0713145108e+02 obj_fn
                      8.0444076396e+00 nln_eq_con_1
```

```
 [  1.4702653619e+01  3.2911324639e+01 ] obj_fn gradient
 [  1.5904312809e+01  3.1808625618e+01 ] nln_eq_con_1 gradient




 Majr Minr    Step  Fun  Merit function Norm gZ  Violtn   nZ Penalty Conv
    0    1 0.0E+00    1  9.90366719E+01 1.6E+00 8.0E+00    1 0.0E+00 F FF

<<omission>>

>>>>> Dakota finite difference gradient evaluation for x[2] - h:

------------------------------
Begin Function Evaluation   40
------------------------------
Parameters for function evaluation 40:
                      4.9873894231e+00 H
                      4.0230575428e+00 D

(container container.in.40 container.out.40)

Active response data for function evaluation 40:
Active set vector = { 1 1 }
                      9.8301287596e+01 obj_fn
                     -1.2698647534e-01 nln_eq_con_1


>>>>> Total response returned to iterator:

Active set vector = { 3 3 }
                      9.8432498115e+01 obj_fn
                     -1.2072307876e-09 nln_eq_con_1
 [  1.3157517799e+01  3.2590159401e+01 ] obj_fn gradient
 [  1.2737124438e+01  3.1548877386e+01 ] nln_eq_con_1 gradient


    7    1 1.0E+00    8  9.84324981E+01 7.9E-11 1.2E-09    1 1.4E+00 T TT

 Exit NPSOL - Optimal solution found.

 Final nonlinear objective value =    98.43250

NPSOL exits with INFORM code = 0 (see "Interpretation of output" section
                                  in NPSOL manual)

NOTE: see Fortran device 9 file (fort.9 or ftn09)
      for complete NPSOL iteration history.

<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 40 total (40 new, 0 duplicate)
<<<<< Best parameters          =
                      4.9873894231e+00 H
                      4.0270846274e+00 D
<<<<< Best objective function  =
                      9.8432498115e+01
<<<<< Best constraint values   =
                     -1.2072307876e-09
```

```
<<<<< Best data captured at function evaluation 36
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =      0.07 [parent =      0.07, child =1.38778e-17]
  Total wall clock =     3.402
```

The first block of lines provide a report on the DAKOTA configuration and settings. The lines that follow, down to the line "`Exit NPSOL - Optimal solution found`", contain information about the function evaluations that have been requested by NPSOL and performed by DAKOTA. Evaluations 6 through 39 have been omitted from the listing for brevity.

Following the line "`Begin Function Evaluation 1`", the initial values of the design variables and the initial objective and constraint function evaluations are listed. The values of the design variables are labeled with the tags `H` and `D`, respectively, according to the descriptors to these variables given in the input file, Figure 7.1. The values of the objective function and volume constraint are labeled with the tags `obj_fn` and `nln_eq_con_1`, respectively. Note that the initial design parameters are infeasible since the equality constraint is violated ($\neq 0$). However, the numerical optimizer has the capability to find a design that is both feasible and optimal for this example. Between the design variables and response values, the content of the system call to the simulator is displayed as "`(container container.in.1 container.out.1)`", with `container` being the name of the simulator and `container.in.1` and `container.out.1` being the names of the parameters and results files, respectively.

Just preceding the output of the objective and constraint function values is the line "`Active set vector = {1 1}`". The active set vector indicates the types of data that are required from the simulator for the objective and constraint functions, and values of "`1`" indicate that the simulator must return values for these functions (gradient and Hessian data are not required). For more information on the active set vector, see Section 4.7.

Since finite difference gradients have been specified, DAKOTA computes their values by making additional function evaluation requests to the simulator at perturbed parameter values. Examples of the gradient-related function evaluations have been included in the sample output, beginning with the line that reads "`>>>>> Dakota finite difference evaluation for x[1] + h:`". The resulting finite difference gradients are listed after function evaluation 5 beginning with the line "`>>>>> Total response returned to iterator:`". Here, another active set vector is displayed in the DAKOTA output file. The line "`Active set vector = { 3 3 }`" indicates that the total response resulting from the finite differencing contains function values and gradients.

The final lines of the DAKOTA output, beginning with the line "`<<<<< Iterator npsol_sqp completed`", summarize the results of the optimization study. The best values of the optimization parameters, objective function, and volume constraint are presented along with the function evaluation number where they occurred, total function evaluation counts, and a timing summary. In the end, the objective function has been minimized and the equality constraint has been satisfied (driven to zero within the constraint tolerance).

The DAKOTA results are intermixed with iteration information from the NPSOL library. The lines with the heading "`Majr Minr Step Fun Merit function Norm gZ Violtn nZ Penalty Conv`" come from Fortran write statements within NPSOL. The output is mixed since both DAKOTA and NPSOL are writing to the same standard output stream. The relative locations of these output contributions can vary depending on the specifics of output buffering and flushing on a particular platform and depending on whether or not the standard output is being redirected to a file. In some cases, output from the optimization library may appear on each iteration (as in this example), and in other cases, it may appear at the end of the DAKOTA output. Finally, a more detailed summary of the NPSOL iterations is written to the Fortran device 9 file (e.g., `fort.9` or `ftn09`).

```
  % eval_id             H             D        obj_fn   nln_eq_con_1
          1           4.5           4.5   107.1314511     8.04440764
          2   5.801246882   3.596476363   94.33737399   -4.591036449
          3   5.197920021   3.923577478   97.77972141  -0.6780884643
          4   4.932877133   4.044776217   98.28930567  -0.1410680155
          5   4.989328734   4.026133158    98.4270019 -0.005324669423
          6   4.987494493   4.027041977   98.43249058 -7.305673455e-06
          7   4.987391669    4.02708372    98.4324981 -1.981308363e-08
          8   4.987389423   4.027084627   98.43249811 -1.207230788e-09
```

Figure 7.2: DAKOTA's tabular output file showing the iteration history of the "container" optimization problem.

7.3 Tabular Output Data

DAKOTA has the capability to print the iteration history in tabular form to a file. The keyword `tabular_graphics_data` needs to be included in the strategy specification (see Figure 7.1). The primary intent of this capability is to facilitate the transfer of DAKOTA's iteration history data to an external mathematical analysis and/or graphics plotting package. Any evaluations from DAKOTA's internal finite differencing are suppressed, which leads to better data visualizations. This suppression of lower level data is consistent with the data that is sent to the graphics windows, as described in Section 7.4. If this data suppression is undesirable, Section 18.2.3 describes an approach where every function evaluation, even the ones from finite differencing, can be saved to a file in tabular format.

The default file name for the tabular output data is "`dakota_tabular.dat`" and the output from the "container" optimization problem is shown in Figure 7.2. This file contains the complete history of data requests from NPSOL (8 requests map into a total of 40 function evaluations when including the central finite differencing). The first column is the data request number, the second and third columns are the design parameter values (labeled in the example as "`H`" and "`D`"), the fourth column is the objective function (labeled "`obj_fn`"), and the fifth column is the nonlinear equality constraint (labeled "`nln_eq_con_1`").

7.4 Graphics Output

Graphics capabilities are available for monitoring the progress of an iterative study. The graphics option is invoked by adding the `graphics` flag in the strategy specification of the DAKOTA input file (see Figure 7.1). The graphics display the values of each response function (e.g., objective and constraint functions) and each parameter for the function evaluations in the study. As for the tabular output described in Section 7.3, internal finite difference evaluations are suppressed in order to omit this clutter from the graphics. Figure 7.3 shows the optimization iteration history for the container example.

If DAKOTA is executed on a remote machine, the DISPLAY variable in the user's UNIX environment [33] may need to be set to the local machine in order to display the graphics window. The scroll bars which are located on each graph below the x-axis and next to the y-axis may be operated by dragging on the bars or pressing the arrows, both of which result in expansion/contraction of the axis scale. Clicking on the options button ("Opt") allows the user to plot the values of the vertical axis using a logarithmic scale so long as all of these values are greater than zero.

In addition to these two-dimensional iteration history plots, three-dimensional surface plots can be generated when using response surface methods in combination with the graphics keyword. This feature is currently available only if there are two parameters in the problem. One common use of response surface methods is in the `surrogate_based_opt` strategy (see Section 13.7), for which a sample specification follows:

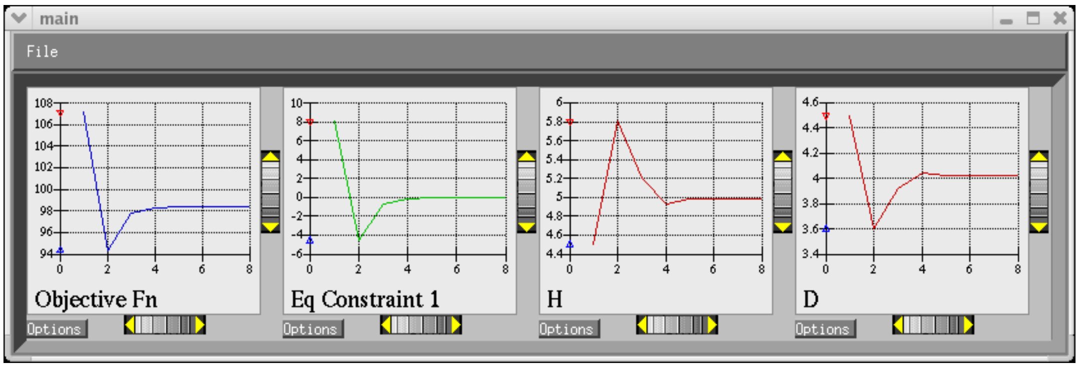


Figure 7.3: DAKOTA output for "container" problem showing history of an objective function, an equality constraint, and two variables.

```
strategy,                                            \
        surrogate_based_opt                          \
        graphics                                     \
        opt_method='NLP'                             \
        trust_region                                 \
          initial_size = 0.10                        \
          contraction_factor = 0.50                  \
          expansion_factor   = 1.50
```

When DAKOTA is executed, a 3-D surface plot is automatically spawned (Figure 7.4 shows an example from optimization of the Rosenbrock problem). The creation of the 3-D surface plot pauses the advance of the optimization algorithm. To continue progress, click the right mouse button or hit return while the mouse cursor is in the 3D graphics window.

The 3D graphics from the PLplot library have a dependency on external font files. If the 3D graphics fail with a message similar to:

```
Cannot open library file: plstnd5.fnt
lib dir="<...some_path...>"
*** PLPLOT ERROR ***
Unable to open font file
Program aborted
```

then the solution is to locate the font files that came with your DAKOTA installation and set the `$PLPLOT_LIB` environment variable to point to them, e.g.:

```
setenv PLPLOT_LIB /home/<user_name>/Dakota/bin
```

7.5 Error Messages Output

A variety of error messages are printed by DAKOTA in the event that an error is detected in the input specification. Some of the more common input errors, and the associated error messages, are described below.

One common mistake is the omission of the continuation symbol "\" when continuing the specifications in a keyword block across multiple lines. When a continuation symbol is omitted, the keyword block is truncated at the point of the omission (by the newline that is not escaped). If this truncation causes loss of a required input, then an error message similar to the following will result:

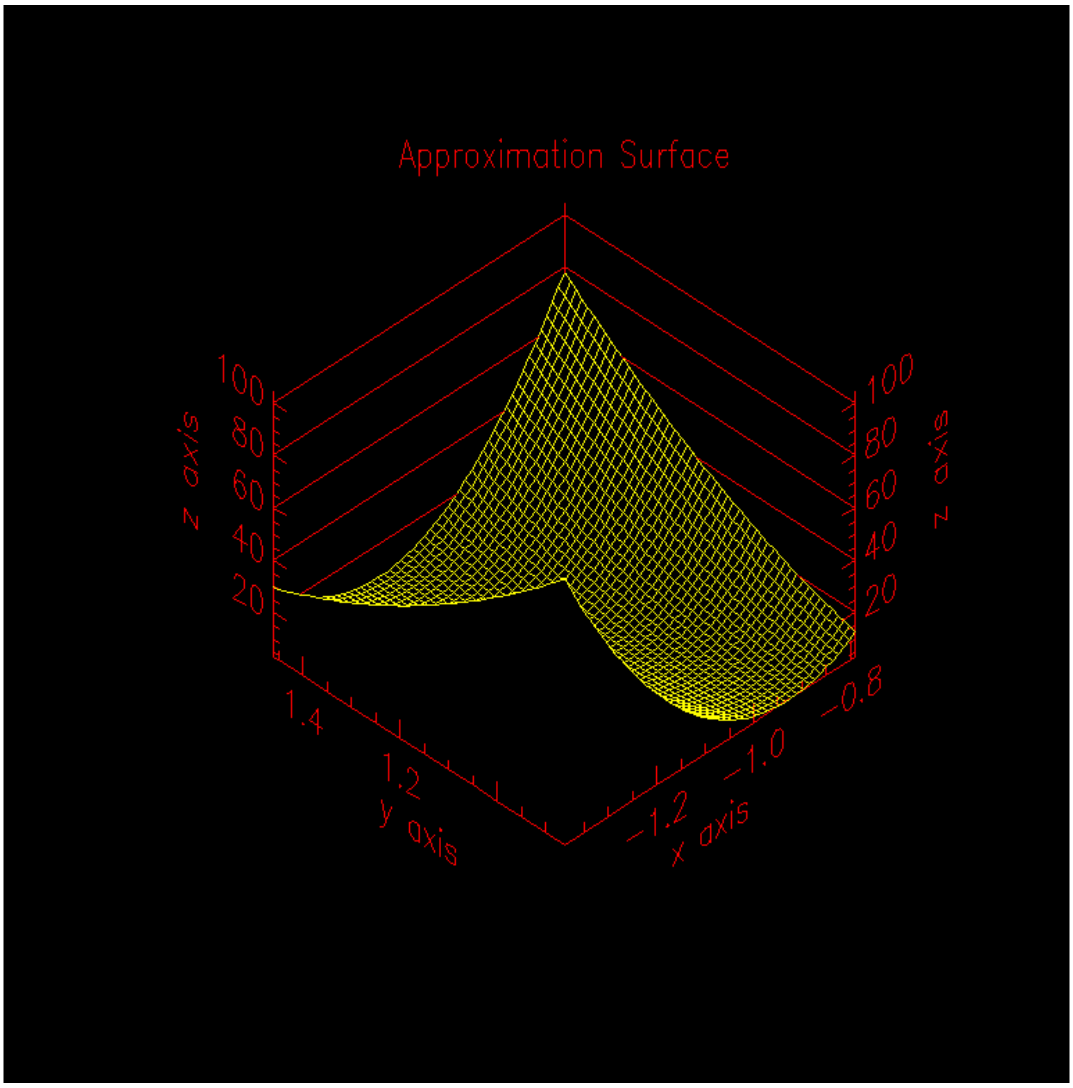


Figure 7.4: An example of the 3-D surface plotting that is available for surrogate-based optimization with two design parameters.

```
    Error: Expected required identifier for keyword
           'responses'.
```

If the truncation results in omission of inputs that are optional, then the parser will still detect a syntax error in the trailing specification that has been disconnected from its keyword block. This error will result in a message similar to the following:

```
    Parser detected syntax error at line 10.  Unrecognized statement.
            Did you forget to escape a newline?
```

Incorrectly spelled specifications will result in error messages of the form:

```
    Parser detected syntax error at line 35.  Unrecognized statement.
```

The input parser catches syntax errors, but not logic errors. The fact that certain input combinations are erroneous must be detected after parsing, at object construction time. For example, if a `no_gradients` specification for a response data set is combined with selection of a gradient-based optimization method, then this error must be detected during set-up of the optimizer (see last two lines of the text listing):

```
    Running MPI executable in serial mode.
    Writing new restart file dakota.rst
    Constructing Single Method Strategy...
    methodName = dot_mmfd
    gradientType = none
    hessianType = none
    DOT Method = 1
    DOT optimization type = minimize
    DOT print control = 7
    Error: gradientType = none is invalid with DOT.
    Please select numerical, analytic, or mixed gradients.
```

Another common mistake involves a mismatch between the amount of data expected on a function evaluation and the data returned by the user's simulation code or driver. The available response data is specified in the responses keyword block, and the subset of this data needed for a particular evaluation is managed by the active set vector. For example, if DAKOTA expects function values and gradients to be returned (as indicated by an active set vector containing 3's), but the user's simulation code only returns function values, then the following error message is generated:

```
    At EOF: insufficient data for functionGradient 1
```

Unfortunately, descriptive error messages are not available for all possible failure modes of DAKOTA. If you encounter core dumps, segmentation faults, or other failures, please report the problem to dakota@sandia.gov.

Chapter 8

Parameter Study Capabilities

8.1 Overview

Parameter study methods in the DAKOTA toolkit involve the computation of response data sets at a selection of points in the parameter space. These response data sets are not linked to any specific interpretation, so they may consist of any allowable specification from the responses keyword block, i.e., objective and constraint functions, least squares terms and constraints, or generic response functions. This allows the use of parameter studies in direct coordination with optimization, least squares, and uncertainty quantification studies without significant modification to the input file. In addition, response data sets are not restricted to function values only; gradients and Hessians of the response functions can also be catalogued by the parameter study. This allows for several different approaches to "sensitivity analysis": (1) the variation of function values over parameter ranges provides a global assessment as to the sensitivity of the functions to the parameters, (2) derivative information can be computed numerically, provided analytically by the simulator, or both (mixed gradients) in directly determining local sensitivity information at a point in parameter space, and (3) the global and local assessments can be combined to investigate the variation of derivative quantities through the parameter space by computing sensitivity information at multiple points.

In addition to sensitivity analysis applications, parameter studies can be used for investigating nonsmoothness in simulation response variations (so that models can be refined or finite difference step sizes can be selected for computing numerical gradients), interrogating problem areas in the parameter space, or performing simulation code verification (verifying simulation robustness) through parameter ranges of interest. A parameter study can also be used in coordination with optimization methods as either a pre-processor (to identify a good starting point) or a post-processor (for post-optimality analysis).

Parameter study methods will iterate any combination of **continuous** design, uncertain, and continuous state variables into any set of responses (any function, gradient, and Hessian definition). Parameter studies draw no distinction between the different types of continuous variables (design, uncertain, or state) and the different types of response functions. They simply pass all of the variables defined in the variables specification into the interface, from which they expect to retrieve all of the responses defined in the responses specification. As described in Section 6.3, when gradient and/or Hessian information is being catalogued in the parameter study, it is assumed that derivative components will be computed with respect to all of the *continuous* variables (continuous design, uncertain, and continuous state variables) specified. Note that if you have a parameter study that you wish to perform on discrete variables, you need to redefine them as continuous and perform the parameter study with a step length equal to `1.0` or an integer multiple equivalent.

DAKOTA currently supports four types of parameter studies. Vector parameter studies compute response data sets at selected intervals along a one-dimensional vector in parameter space. List parameter studies compute response data sets at a list of points in parameter space, defined by the user. A centered parameter study computes multiple coordinate-based parameter studies, one per parameter, centered about the initial parameter values. A multidimensional parameter study computes response data sets for an n-dimensional hypergrid of points. More detail on these types of parameter studies is found in Sections 8.2 through 8.5 below.

8.1.1 Initial Values

The vector and centered parameter studies use the initial values of the variables from the variables keyword block as the starting point and the central point of the parameter studies, respectively. In the case of design variables, the `initial_point` is used. In the case of state variables, the `initial_state` is used. In the case of uncertain variables, initial values for variables with normal, lognormal, uniform, loguniform, weibull, and histogram probability distributions are the mean, mean, mid-point between bounds, mid-point between bounds, beta parameter, and bin/point lower bound, respectively. The initial values for variables with triangular, beta, gamma, gumbel, and frechet distributions are the mode, mean, alpha parameter/beta parameter, mean, and mean, respectively. These starting values for design, uncertain, and state variables are referenced repeatedly in the following sections using the identifier “Initial Values.”

8.2 Vector Parameter Study

The vector parameter study computes response data sets at selected intervals along a one-dimensional vector in parameter space. This capability encompasses both single-coordinate parameter studies (to study the effect of a single variable on a response set) as well as multiple coordinate vector studies (to investigate the response variations along some n-dimensional vector; e.g., to investigate a search direction failure). In addition to these uses, this capability is used recursively within the implementation of the multidimensional parameter study.

DAKOTA’s vector parameter study includes three possible specification formulations which are used in conjunction with the Initial Values (see Section 8.1.1) to define the vector and steps of the parameter study:

```
final_point (vector of reals) and step_length (real)
final_point (vector of reals) and num_steps (integer)
step_vector (vector of reals) and num_steps (integer)
```

In each of these three cases, the Initial Values are used as the parameter study starting point and the specification selected from the three above defines the orientation and length of the vector as well as the increments to be evaluated along the vector. Several examples starting from Initial Values of `1.0, 1.0, 1.0` are included below:

`final_point = 1.0, 2.0, 1.0` and `step_length = .4`:

```
Parameters for function evaluation 1:
                      1.0000000000e+00 d1
                      1.0000000000e+00 d2
                      1.0000000000e+00 d3
Parameters for function evaluation 2:
                      1.0000000000e+00 d1
                      1.4000000000e+00 d2
                      1.0000000000e+00 d3
Parameters for function evaluation 3:
                      1.0000000000e+00 d1
                      1.8000000000e+00 d2
                      1.0000000000e+00 d3
```

`final_point = 2.0, 2.0, 2.0` and `step_length = .4` (note that `step_length` defines Cartesian distance of the step and the steps continue up to but not past the `final_point`):

```
    Parameters for function evaluation 1:
                          1.0000000000e+00 d1
                          1.0000000000e+00 d2
                          1.0000000000e+00 d3
    Parameters for function evaluation 2:
                          1.2309401077e+00 d1
                          1.2309401077e+00 d2
                          1.2309401077e+00 d3
    Parameters for function evaluation 3:
                          1.4618802154e+00 d1
                          1.4618802154e+00 d2
                          1.4618802154e+00 d3
    Parameters for function evaluation 4:
                          1.6928203230e+00 d1
                          1.6928203230e+00 d2
                          1.6928203230e+00 d3
    Parameters for function evaluation 5:
                          1.9237604307e+00 d1
                          1.9237604307e+00 d2
                          1.9237604307e+00 d3
```

`final_point = 2.0, 2.0, 2.0` and `num_steps = 4`:

```
    Parameters for function evaluation 1:
                          1.0000000000e+00 d1
                          1.0000000000e+00 d2
                          1.0000000000e+00 d3
    Parameters for function evaluation 2:
                          1.2500000000e+00 d1
                          1.2500000000e+00 d2
                          1.2500000000e+00 d3
    Parameters for function evaluation 3:
                          1.5000000000e+00 d1
                          1.5000000000e+00 d2
                          1.5000000000e+00 d3
    Parameters for function evaluation 4:
                          1.7500000000e+00 d1
                          1.7500000000e+00 d2
                          1.7500000000e+00 d3
    Parameters for function evaluation 5:
                          2.0000000000e+00 d1
                          2.0000000000e+00 d2
                          2.0000000000e+00 d3
```

`step_vector = .1, .1, .1` and `num_steps = 4`:

```
    Parameters for function evaluation 1:
                          1.0000000000e+00 d1
                          1.0000000000e+00 d2
                          1.0000000000e+00 d3
    Parameters for function evaluation 2:
                          1.1000000000e+00 d1
                          1.1000000000e+00 d2
                          1.1000000000e+00 d3
    Parameters for function evaluation 3:
                          1.2000000000e+00 d1
                          1.2000000000e+00 d2
                          1.2000000000e+00 d3
```

```
Parameters for function evaluation 4:
                      1.3000000000e+00 d1
                      1.3000000000e+00 d2
                      1.3000000000e+00 d3
Parameters for function evaluation 5:
                      1.4000000000e+00 d1
                      1.4000000000e+00 d2
                      1.4000000000e+00 d3
```

8.3 List Parameter Study

The list parameter study computes response data sets at selected points in parameter space. These points are explicitly specified by the user and are not confined to lie on any line or surface. Thus, this parameter study provides a general facility that supports the case where the desired set of points to evaluate does not fit the prescribed structure of the vector, centered, or multidimensional parameter studies.

The user input consists of a `list_of_points` specification which lists the requested parameter sets in succession. The list parameter study simply performs a simulation for the first parameter set (the first n entries in the list), followed by a simulation for the next parameter set (the next n entries), and so on, until the list of points has been exhausted. Since the Initial Values will not be used, they need not be specified.

An example specification which would result in the same parameter sets as in the first example in Section 8.2 would be:

```
list_of_points = 1.0, 1.0, 1.0, 1.0, 1.4, 1.0, 1.0, 1.8, 1.0
```

8.4 Centered Parameter Study

The centered parameter study executes multiple coordinate-based parameter studies, one per parameter, centered about the specified Initial Values. This is useful for investigation of function contours in the vicinity of a specific point. For example, after computing an optimum design, this capability could be used for post-optimality analysis in verifying that the computed solution is actually at a minimum or constraint boundary and in investigating the shape of this minimum or constraint boundary.

This method requires `percent_delta` (real) and `deltas_per_variable` (integer) specifications, where the former specifies the size of the increments in percent and the latter specifies the number of increments per variable in each of the plus and minus directions.

For example, with Initial Values of `1.0, 1.0`, a `percent_delta` of `10.0`, and a `deltas_per_variable` of `2`, the center point is evaluated followed by four function evaluations (two minus deltas and two plus deltas) per variable:

```
Parameters for function evaluation 1:
                      1.0000000000e+00 cdv_1
                      1.0000000000e+00 cdv_2
Parameters for function evaluation 2:
                      8.0000000000e-01 cdv_1
                      1.0000000000e+00 cdv_2
Parameters for function evaluation 3:
                      9.0000000000e-01 cdv_1
                      1.0000000000e+00 cdv_2
Parameters for function evaluation 4:
                      1.1000000000e+00 cdv_1
                      1.0000000000e+00 cdv_2
Parameters for function evaluation 5:
```

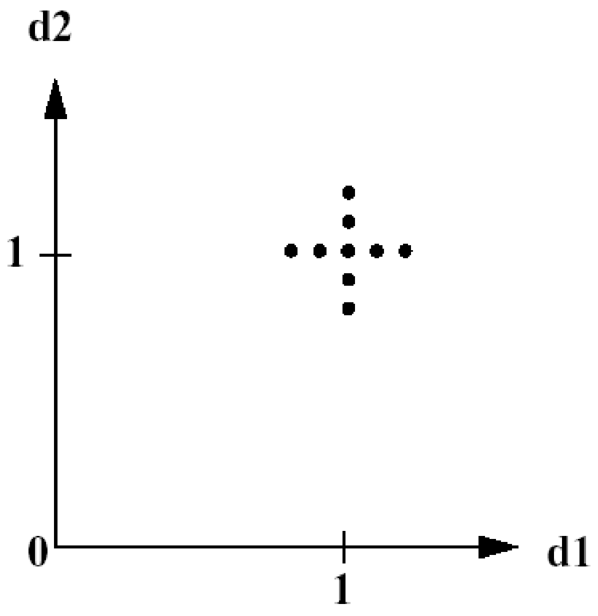


Figure 8.1: Example centered parameter study.

```
                      1.2000000000e+00 cdv_1
                      1.0000000000e+00 cdv_2
Parameters for function evaluation 6:
                      1.0000000000e+00 cdv_1
                      8.0000000000e-01 cdv_2
Parameters for function evaluation 7:
                      1.0000000000e+00 cdv_1
                      9.0000000000e-01 cdv_2
Parameters for function evaluation 8:
                      1.0000000000e+00 cdv_1
                      1.1000000000e+00 cdv_2
Parameters for function evaluation 9:
                      1.0000000000e+00 cdv_1
                      1.2000000000e+00 cdv_2
```

This set of points in parameter space is depicted in Figure 8.1.

If the Initial Values for the centered parameter study are very small or equal to zero, the study will substitute a default step size. This is necessary due to the relative nature of the `percent_delta` specification.

8.5 Multidimensional Parameter Study

The multidimensional parameter study computes response data sets for an n-dimensional hypergrid of points. Each continuous variable is partitioned into equally spaced intervals between its upper and lower bounds, and each combination of the values defined by these partitions is evaluated. The number of function evaluations performed in the study is:

$$\prod_{i=1}^{n}(\texttt{partitions}_i + 1) \tag{8.1}$$

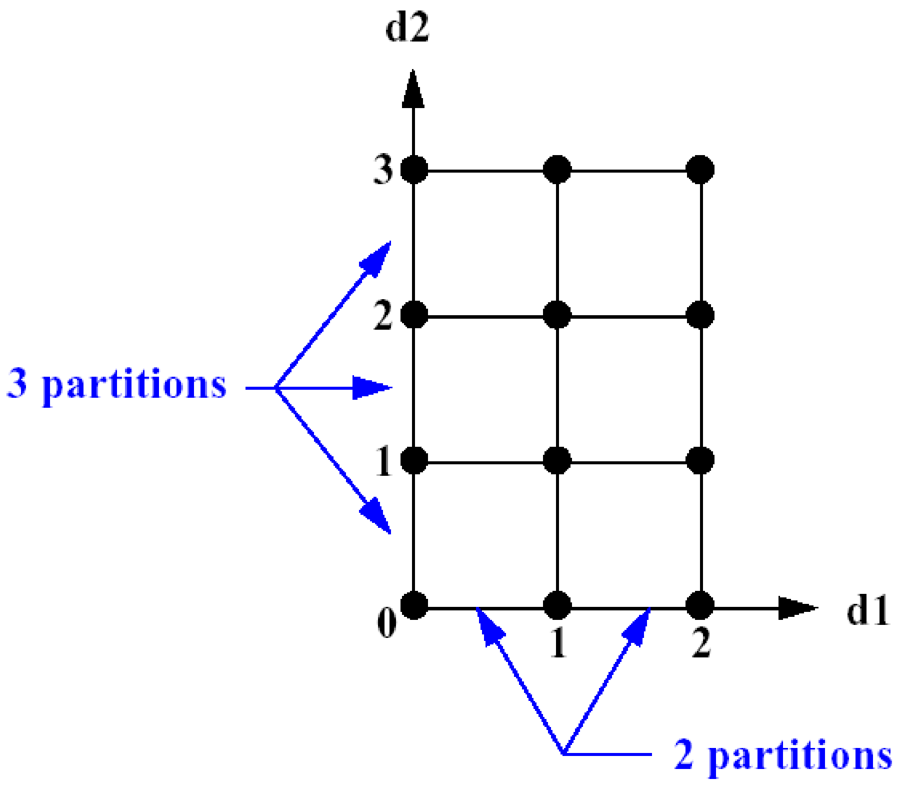


Figure 8.2: Example multidimensional parameter study

The partitions information is specified using the `partitions` specification, which provides an integer list of the number of partitions for each continuous variable (i.e., `partitions`$_i$). Since the Initial Values will not be used, they need not be specified.

In a two variable example problem with `d1` ∈ `[0,2]` and `d2` ∈ `[0,3]` (as defined by the upper and lower bounds specified in the variables specification) and with `partitions = 2,3`, the interval `[0,2]` is divided into two equal-sized partitions and the interval `[0,3]` is divided into three equal-sized partitions. This two-dimensional grid, shown in Figure 8.2,

would result in the following twelve function evaluations:

```
Parameters for function evaluation 1:
                      0.0000000000e+00 d1
                      0.0000000000e+00 d2
Parameters for function evaluation 2:
                      1.0000000000e+00 d1
                      0.0000000000e+00 d2
Parameters for function evaluation 3:
                      2.0000000000e+00 d1
                      0.0000000000e+00 d2
Parameters for function evaluation 4:
                      0.0000000000e+00 d1
                      1.0000000000e+00 d2
Parameters for function evaluation 5:
                      1.0000000000e+00 d1
                      1.0000000000e+00 d2
Parameters for function evaluation 6:
                      2.0000000000e+00 d1
```

```
                          1.0000000000e+00 d2
    Parameters for function evaluation 7:
                          0.0000000000e+00 d1
                          2.0000000000e+00 d2
    Parameters for function evaluation 8:
                          1.0000000000e+00 d1
                          2.0000000000e+00 d2
    Parameters for function evaluation 9:
                          2.0000000000e+00 d1
                          2.0000000000e+00 d2
    Parameters for function evaluation 10:
                          0.0000000000e+00 d1
                          3.0000000000e+00 d2
    Parameters for function evaluation 11:
                          1.0000000000e+00 d1
                          3.0000000000e+00 d2
    Parameters for function evaluation 12:
                          2.0000000000e+00 d1
                          3.0000000000e+00 d2
```

Chapter 9

Design of Experiments and Sampling Methods

9.1 Overview

DAKOTA contains three software packages that can be used for sampling and design of experiments: LHS (Latin hypercube sampling), DDACE (distributed design and analysis for computer experiments), and FSU-Dace (Florida State University's Design and Analysis of Computer Experiments package). LHS [71] is a general-purpose sampling package developed at Sandia that has been used by the DOE national labs for several decades. DDACE is a more recent package for computer experiments that is under development by staff at Sandia Labs [64]. DDACE provides the capability for generating orthogonal arrays, Box-Behnken designs, Central Composite designs, and random designs. DDACE is available under a GNU Lesser General Public License and is distributed with DAKOTA. The FSUDace package provides the following sampling techniques: quasi-Monte Carlo sampling based on Halton or Hammersley sequences, and Centroidal Voronoi Tessellation. FSUDace is available under a GNU Lesser General Public License and is distributed with DAKOTA.

This chapter focuses on DDACE and FSUDace, with the primary goal of designing computer experiments. Latin Hypercube Sampling, used in uncertainty quantification, is discussed in Section 10.2. The differences between sampling used in design of experiments and sampling used in uncertainty quantification is discussed in more detail in the following paragraphs. In brief, we consider design of experiment methods to generate sets of uniform random variables on the interval $[0, 1]$. These sets are mapped to the lower/upper bounds of the problem variables and then the response functions are evaluated at the sample input points with the goal of characterizing the behavior of the response functions over the input parameter ranges of interest. Uncertainty quantification via LHS sampling, in contrast, involves characterizing the uncertain input variables with probability distributions such as normal, Weibull, triangular, etc., sampling from the input distributions, and propagating the input uncertainties to obtain a cumulative distribution function on the output. There is significant overlap between design of experiments and sampling. Often, both techniques can be used to obtain similar results about the behavior of the response functions and about the relative importance of the input variables.

9.2 Design of Computer Experiments

Computer experiments are often different from physical experiments, such as those performed in agriculture, manufacturing, or biology. In physical experiments, one often applies the same *treatment* or *factor level* in an experiment several times to get an understanding of the variability of the output when that treatment is applied. For example, in an agricultural experiment, several fields (e.g., 8) may be subject to a low level of fertilizer and the same number of fields may be subject to a high level of fertilizer to see if the amount of fertilizer has a significant effect on crop output. In addition, one is often interested in the variability of the output within a treatment group: is the variability of the crop yields in the low fertilizer group much higher than that in the high fertilizer group, or not?

In physical experiments, the process we are trying to examine is stochastic: that is, the same treatment may result in different outcomes. By contrast, in computer experiments, often we have a deterministic code. If we run the code with a particular set of input parameters, the code will always produce the same output. There certainly are stochastic codes, but the main focus of computer experimentation has been on deterministic codes. Thus, in computer experiments we often do not have the need to do replicates (running the code with the exact same input parameters several times to see differences in outputs). Instead, a major concern in computer experiments is to create an experimental design which can sample a high-dimensional space in a representative way with a minimum number of samples. The number of factors or parameters that we wish to explore in computer experiments is usually much higher than physical experiments. In physical experiments, one may be interesting in varying a few parameters, usually five or less, while in computer experiments we often have dozens of parameters of interest. Choosing the levels of these parameters so that the samples adequately explore the input space is a challenging problem. There are many experimental designs and sampling methods which address the issue of adequate and representative sample selection.

There are many goals of running a computer experiment: one may want to explore the input domain or the design space and get a better understanding of the range in the outputs for a particular domain. Another objective is to determine which inputs have the most influence on the output, or how changes in the inputs change the output. This is usually called *sensitivity analysis*. Another goal is to compare the relative importance of model input uncertainties on the uncertainty in the model outputs, *uncertainty analysis*. Yet another goal is to use the sampled inputs points and their corresponding output to create a *response surface approximation* for the computer code. The response surface approximation (e.g., a polynomial regression model, a kriging model, a neural net) can then be used to emulate the computer code. Constructing a response surface approximation is particularly important for applications where running a computational model is extremely expensive: the computer model may take 10 or 20 hours to run on a high performance machine, whereas the response surface model may only take a few seconds. Thus, one often optimizes the response surface model or uses it within a framework such as surrogate-based optimization. Response surface models are also valuable in cases where the gradient (first derivative) and/or Hessian (second derivative) information required by optimization techniques are either not available, expensive to compute, or inaccurate because the derivatives are poorly approximated or the function evaluation is itself noisy due to roundoff errors. Furthermore, many optimization methods require a good initial point to ensure fast convergence or to converge to good solutions (e.g. for problems with multiple local minima). Under these circumstances, a good design of computer experiment framework coupled with response surface approximations can offer great advantages.

In addition to the sensitivity analysis, uncertainty analysis, and response surface modeling mentioned above, we also may want to do *uncertainty quantification* on a computer model. Uncertainty quantification (UQ) refers to taking a particular set of distributions on the inputs, and propagating them through the model to obtain a distribution on the outputs. For example, if input parameter A follows a normal with mean 5 and variance 1, the computer produces a random draw from that distribution. If input parameter B follows a weibull distribution with alpha = 0.5 and beta = 1, the computer produces a random draw from that distribution. When all of the uncertain variables have samples drawn from their input distributions, we run the model with the sampled values as inputs. We do this repeatedly to build up a distribution of outputs. We can then use the cumulative distribution function of the output to ask questions such as: what is the probability that the output is greater than 10? What is the 99th percentile of the output?

Note that sampling-based uncertainty quantification and design of computer experiments are very similar. THERE IS SIGNIFICANT OVERLAP in the purpose and methods used for UQ and for DACE. We have attempted to delineate the differences within DAKOTA as follows: we use the two methods, DDACE and FSUDACE, primarily for design of experiments, where we are interested in understanding the main effects of parameters and where we want to sample over an input domain to obtain values for constructing a response surface. We use the nondeterministic sampling methods (`nond_sampling`) for uncertainty quantification, where we are propagating specific input distributions and interested in obtaining (for example) a cumulative distribution function on the output. If you have a problem where you have no distributional information, we recommend starting with a design of experiments approach. Note that DDACE and FSUDACE currently do NOT support distributional information: they take an upper and lower bound for each uncertain input variable and sample within that. The uncertainty quantification methods in `nond_sampling` (primarily Latin Hypercube sampling) offer the capability to sample from many distributional types. The distinction between UQ and DACE is somewhat arbitrary: both approaches often can yield insight about important parameters and both can determine sample points for response surface approximations.

9.3 DDACE Background

The DACE package includes both classical design of experiments methods [64] and stochastic sampling methods. The classical design of experiments methods in DDACE are central composite design (CCD) and Box-Behnken (BB) sampling. A grid-based sampling method also is available. The stochastic methods are orthogonal array sampling [47], Monte Carlo (random) sampling, and Latin hypercube sampling. Note that the DDACE version available through the DAKOTA interface only supports uniform distributions. DDACE does not currently support enforcement of user-specified correlation structure among the variables.

The sampling methods in DDACE can be used alone or in conjunction with other methods. For example, DDACE sampling can be used with both the surrogate-based optimization strategy and the optimization under uncertainty strategy. See Figure 13.9 for an example of how the DDACE settings are used in DAKOTA.

More information on DDACE is available on the web at: http://csmr.ca.sandia.gov/projects/ddace

The following sections provide more detail about the sampling methods available for design of experiments in DDACE.

9.3.1 Central Composite Design

A Box-Wilson Central Composite Design, commonly called a central composite design (CCD), contains an embedded factorial or fractional factorial design with center points that is augmented with a group of 'star points' that allow estimation of curvature. If the distance from the center of the design space to a factorial point is ± 1 unit for each factor, the distance from the center of the design space to a star point is $\pm\alpha$ with $\mid \alpha \mid > 1$. The precise value of depends on certain properties desired for the design and on the number of factors involved. The CCD design is specified in DAKOTA with the method command `ddace central_composite`.

As an example, with a two input variables or factors, each having two levels, the factorial design is shown in Table 9.1.

Table 9.1: Simple Factorial Design

Input 1	Input 2
-1	-1
-1	+1
+1	-1
+1	+1

With a CCD, the design above would be augmented with the following points, if α = 1.3:

Table 9.2: Additional Points to make the factorial design a CCD

Input 1	Input 2
0	+1.3
0	-1.3
1.3	0
-1.3	0
0	0

These points define a circle around the original factorial design.

Note that the number of samples points specified in a CCD,`samples`, is a function of the number of variables in the problem:

$$samples = 1 + 2 * NumVar + 2^{NumVar}$$

9.3.2 Box-Behnken Design

The Box-Behnken design is similar to a Central Composite design, with some differences. The Box-Behnken design is a quadratic design in that it does not contain an embedded factorial or fractional factorial design. In this design the treatment combinations are at the midpoints of edges of the process space and at the center, as compared with CCD designs where the extra points are placed at 'star points' on a circle outside of the process space. Box-Behken designs are rotatable (or near rotatable) and require 3 levels of each factor. The designs have limited capability for orthogonal blocking compared to the central composite designs. Box-Behnken requires fewer runs than CCD for 3 factors, but this advantage goes away as the number of factors increases. The Box-Behnken design is specified in DAKOTA with the method command `ddace box_behnken`.

Note that the number of samples points specified in a Box-Behnken design, `samples`, is a function of the number of variables in the problem:

$$samples = 1 + 4 * NumVar + (NumVar - 1)/2$$

9.3.3 Orthogonal Array Designs

Orthogonal array (OA) sampling was independently considered by Owen and Tang. An orthogonal array sample can be described as an 4-tuple (m, n, s, r), where m is the number of sample points, n is the number of input variables, s is the number of symbols, and r is the strength of the orthogonal array. The number of sample points, m, must be a multiple of the number of symbols, s. The number of symbols refers to the number of levels per input variable. The strength refers to the number of columns where we are guaranteed to see all the possibilities an equal number of times.

For example, Table 9.3 shows an orthogonal array of strength 2 for m = 8, with 7 variables:

If one picks any two columns, say the first and the third, note that each of the four possible rows we might see there, 0 0, 0 1, 1 0, 1 1, appears exactly the same number of times, twice in this case.

DDACE creates orthogonal arrays of strength 2. Further, the OAs generated by DDACE do not treat the factor levels as one fixed value (0 or 1 in the above example). Instead, once a level for a variable is determined in the array, DDACE samples a random variable from within that level. The orthogonal array design is specified in DAKOTA with the method command `ddace oas`.

Table 9.3: Orthogonal Array for Seven Variables

Input 1	Input 2	Input 3	Input 4	Input 5	Input 6	Input 7
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	0	1	0
1	1	0	0	1	1	0
1	1	0	1	0	0	1

The orthogonal array method in DDACE is the only method that allows for the calculation of main effects, specified with the command `main_effects`. Main effects is a sensitivity analysis method which identifies the input variables that have the most influence on the output. In main effects, the idea is to look at the mean of the response function when variable A (for example) is at level 1 vs. when variable A is at level 2 or level 3. If these mean responses of the output are statistically significantly different at different levels of variable A, this is an indication that variable A has a significant effect on the response. The orthogonality of the columns is critical in performing main effects analysis, since the column orthogonality means that the effects of the other variables 'cancel out' when looking at the overall effect from one variable at its different levels. There are ways of developing orthogonal arrays to calculate higher order interactions, such as two-way interactions (what is the influence of Variable A * Variable B on the output?), but this is not available in DDACE currently. At present, one way interactions are supported in the calculation of orthogonal array main effects within DDACE.

9.3.4 Grid Design

In a grid design, a grid is placed over the input variable space. This is very similar to a multi-dimensional parameter study where the samples are taken over a set of partitions on each variable (see Section 8.5). The main difference is that in grid sampling, a small random perturbation is added to each sample value so that the grid points are not on a perfect grid. This is done to help capture certain features in the output such as periodic functions. A purely structured grid, with the samples exactly on the grid points, has the disadvantage of not being able to capture important features such as periodic functions with relatively high frequency (due to aliasing). Adding a random perturbation to the grid samples helps remedy this problem.

Another disadvantage with grid sampling is that the number of sample points required depends exponentially on the input dimensions. In grid sampling, the number of samples is the number of symbols (grid partitions) raised to the number of variables. For example, if there are 2 variables, each with 5 partitions, the number of samples would be 5^2. In this case, doubling the number of variables squares the sample size. The grid design is specified in DAKOTA with the method command `ddace grid`.

9.3.5 Monte Carlo Design

Monte Carlo designs simply involve pure Monte-Carlo random sampling from uniform distributions between the lower and upper bounds on each of the input variables. Monte Carlo designs, specified by `ddace random`, are a way to generate a set of random samples over an input domain.

9.3.6 LHS Design

DDACE offers the capability to generate Latin Hypercube designs. For more information on Latin Hypercube sampling, see Section 10.2. Note that the version of LHS in DDACE generates uniform samples (uniform between the variable bounds). The version of LHS offered with nondeterministic sampling can generate LHS samples according to a number of distribution types, including normal, lognormal, weibull, beta, etc. To specify the DDACE version of LHS, use the method command `ddace lhs`.

9.4 FSUDace Background

The FSUDace package includes quasi-Monte Carlo sampling methods (Halton and Hammersley sequences) and Centroidal Voronoi Tesselation sampling. All three methods generate sets of uniform random variables on the interval $[0, 1]$. The quasi-Monte Carlo and CVT methods are designed with the goal of low discrepancy. Discrepancy refers to the nonuniformity of the sample points within the unit hypercube. Low discrepancy sequences tend to cover the unit hypercube reasonably uniformly. Quasi-Monte Carlo methods produce low discrepancy sequences, especially if one is interested in the uniformity of projections of the point sets onto lower dimensional faces of the hypercube (usually 1-D: how well do the marginal distributions approximate a uniform?) CVT does very well volumetrically: it spaces the points fairly equally throughout the space, so that the points cover the region and are isotropically distributed with no directional bias in the point placement. There are various measures of volumetric uniformity which take into account the distances between pairs of points, regularity measures, etc. Note that CVT does not produce low-discrepancy sequences in lower dimensions, however: the lower-dimension (such as 1-D) projections of CVT can have high discrepancy.

The quasi-Monte Carlo sequences of Halton and Hammersley are deterministic sequences determined by a set of prime bases. A Halton design is specified in DAKOTA with the method command `fsu_quasi_mc halton`, and the Hammersley design is specified with the command `fsu_quasi_mc hammersley`. For more details about the input specification, see the Reference Manual. CVT points tend to arrange themselves in a pattern of cells that are roughly the same shape. To produce CVT points, an almost arbitrary set of initial points is chosen, and then an internal set of iterations is carried out. These iterations repeatedly replace the current set of sample points by an estimate of the centroids of the corresponding Voronoi subregions [73]. A CVT design is specified in DAKOTA with the method command `fsu_cvt`.

The methods in FSUDace are useful for design of experiments because they provide good coverage of the input space, thus allowing global sensitivity analysis.

9.5 Sensitivity Analysis

Like parameter studies (see Chapter 8), the DACE techniques are useful for characterizing the behavior of the response functions of interest through the parameter ranges of interest. In addition to direct interrogation and visualization of the sampling results, a number of techniques have been developed for assessing the parameters which are most influential in the observed variability in the response functions. One example of this is the well-known technique of scatter plots, in which the set of samples is projected down and plotted against one parameter dimension, for each parameter in turn. Scatter plots with a uniformly distributed cloud of points indicate parameters with little influence on the results, whereas scatter plots with a defined shape to the cloud indicate parameters which are more significant. Related techniques include analysis of variance (ANOVA) [54] and main effects analysis, in which the parameters which have the greatest influence on the results are identified from sampling results. Scatter plots and ANOVA may be accessed through import of DAKOTA tabular results (see Section 7.3) into external statistical analysis programs such as S-plus, Minitab, etc.

Running any of the design of experiments or sampling methods allows the user to save the results in a tabular data file, which then can be read into a spreadsheet or statistical package for further analysis. In addition, we have provided some functions to help determine the most important variables.

We take the definition of uncertainty analysis from [74]: ”The study of how uncertainty in the output of a model can be apportioned to different sources of uncertainty in the model input.”

As a default, DAKOTA provides correlation analyses when running LHS. Correlation tables are printed with the simple, partial, and rank correlations between inputs and outputs. These can be useful to get a quick sense of how correlated the inputs are to each other, and how correlated various outputs are to inputs. The correlation analyses are explained further in Chapter 10.2.

We also have the capability to calculate sensitivity indices through Variance-based Decomposition (VBD). Variance-based decomposition is a way of using sets of samples to understand how the variance of the output behaves, with respect to each input variable. A larger value of the sensitivity index, S_i (`Si` in the DAKOTA output), means that the uncertainty in the input variable i has a larger effect on the variance of the output. More details on the calculations and interpretation of the sensitivity indices can be found in [74]. VBD can be specified for any of the sampling methods using the command `variance_based_decomposition`. Note that VBD is extremely computationally intensive since replicated sets of sample values are evaluated. If the user specified a number of samples, N, and a number of nondeterministic variables, M, variance-based decomposition requires the evaluation of $N(M+2)$ samples. To obtain sensitivity indices that are reasonably accurate, we recommend that N, the number of samples, be at least one hundred and preferably several hundred or thousands. Because of the computational cost, Variance-based decomposition is turned off as a default.

Chapter 10

Nondeterministic Analysis and Uncertainty Quantification

10.1 Overview

DAKOTA contains the DAKOTA/UQ software package for performing nondeterministic analysis. The DAKOTA/UQ package is tightly-woven into the core DAKOTA software and is not available separately. The methods in DAKOTA/UQ have been developed by a group of researchers at Sandia Labs, in conjunction with collaborators in academia [29], [30]. In addition, future extensions to the DDACE package will make it applicable to general UQ problems, which will augment the DAKOTA/UQ capabilities.

Uncertainty quantification methods (also referred to as nondeterministic analysis methods) in the DAKOTA/UQ system involve the computation of probabilistic information about response functions based on sets of simulations taken from the specified probability distributions for uncertain parameters. That is, these methods perform a forward uncertainty propagation in which probability information for input parameters is mapped to probability information for output response functions. The m functions in the DAKOTA response data set are interpreted as m general response functions by the DAKOTA/UQ methods (with no specific interpretation of the functions as for optimization and least squares).

Within the variables specification, uncertain variable descriptions are employed to define the parameter probability distributions (see Section 4.3). The distribution types include: normal (Gaussian), lognormal, uniform, loguniform, weibull, triangular, beta, gamma, gumbel, frechet, interval, and user-defined histogram. All uncertain variables are treated as continuous variables in DAKOTA. Thus, when gradient and/or Hessian information is used in an uncertainty assessment, it is assumed that derivative components will be computed with respect to the *uncertain variables*.

10.2 Sampling Methods

Sampling techniques are selected using the `nond_sampling` method selection. This method generates sets of samples according to the probability distributions of the uncertain variables and maps them into corresponding sets of response functions, where the number of samples is specified by the `samples` integer specification. Means, standard deviations, coefficients of variance (COVs), and 95% confidence intervals are computed for the response functions. Probabilities of occurrence are assessed by comparing the response results against a set of user-supplied thresholds from the `response_thresholds` specification.

Currently, traditional Monte Carlo (MC) and Latin hypercube sampling (LHS) are supported by DAKOTA and are chosen by specifying `sample_type` as `random` or `lhs`. In Monte Carlo sampling, the samples are selected randomly according to the user-specified probability distributions. Latin hypercube sampling is a stratified sampling technique for which the range of each uncertain variable is divided into N_s segments of equal probability, where N_s is the number of samples requested. The relative lengths of the segments are determined by the nature of the specified probability distribution (e.g., uniform has segments of equal width, normal has small segments near the mean and larger segments in the tails). For each of the uncertain variables, a sample is selected randomly from each of these equal probability segments. These N_s values for each of the individual parameters are then combined in a shuffling operation to create a set of N_s parameter vectors with a specified correlation structure. A feature of the resulting sample set is that *every row and column in the hypercube of partitions has exactly one sample*. Since the total number of samples is exactly equal to the number of partitions used for each uncertain variable, an arbitrary number of desired samples is easily accommodated (as compared to less flexible approaches in which the total number of samples is a product or exponential function of the number of intervals for each variable, i.e., many classical design of experiments methods).

Advantages of sampling-based methods include their relatively simple implementation and their independence from the scientific disciplines involved in the analysis. The main drawback of these techniques is the large number of function evaluations needed to generate converged statistics, which can render such an analysis computationally very expensive, if not intractable, for real-world engineering applications. LHS techniques, in general, require fewer samples than traditional Monte Carlo for the same accuracy in statistics, but they still can be prohibitively expensive. For further information on the method and its relationship to other sampling techniques, one is referred to the works by McKay, et al. [49], Iman and Shortencarier [45], and Helton and Davis [43]. Note that under certain monotonicity conditions associated with the function to be sampled, Latin hypercube sampling provides a more accurate estimate of the mean value than does random sampling. That is, given an equal number of samples, the LHS estimate of the mean will have less variance than the mean value obtained through random sampling.

Figure 10.1 demonstrates Latin hypercube sampling on a two-variable parameter space. Here, the range of both parameters, x_1 and x_2, is $[0, 1]$. Also, for this example both x_1 and x_2 have uniform statistical distributions. For Latin hypercube sampling, the range of each parameter is divided into p "bins" of equal probability. For parameters with uniform distributions, this corresponds to partitions of equal size. For n design parameters, this partitioning yields a total of p^n bins in the parameter space. Next, p samples are randomly selected in the parameter space, with the following restrictions: (a) each sample is randomly placed inside a bin, and (b) for all one-dimensional projections of the p samples and bins, there will be one and only one sample in each bin. In a two-dimensional example such as that shown in Figure 10.1, these LHS rules guarantee that only one bin can be selected in each row and column. For $p = 4$, there are four partitions in both x_1 and x_2. This gives a total of 16 bins, of which four will be chosen according to the criteria described above. Note that there is more than one possible arrangement of bins that meet the LHS criteria. The dots in Figure 10.1 represent the four sample sites in this example, where each sample is randomly located in its bin. There is no restriction on the number of bins in the range of each parameter, however, all parameters must have the same number of bins.

The actual algorithm for generating Latin hypercube samples is more complex than indicated by the description given above. For example, the Latin hypercube sampling method implemented in the LHS code [71] takes into account a user-specified correlation structure when selecting the sample sites. For more details on the implementation of the LHS algorithm, see Reference [71].

10.2.1 Uncertainty Quantification Example using Sampling Methods

The following response functions from the Textbook example problem (see Chapter 20):

$$f = (x_1 - 1)^4 + (x_2 - 1)^4$$

$$c_1 = x_1^2 - \frac{1}{2}x_2$$

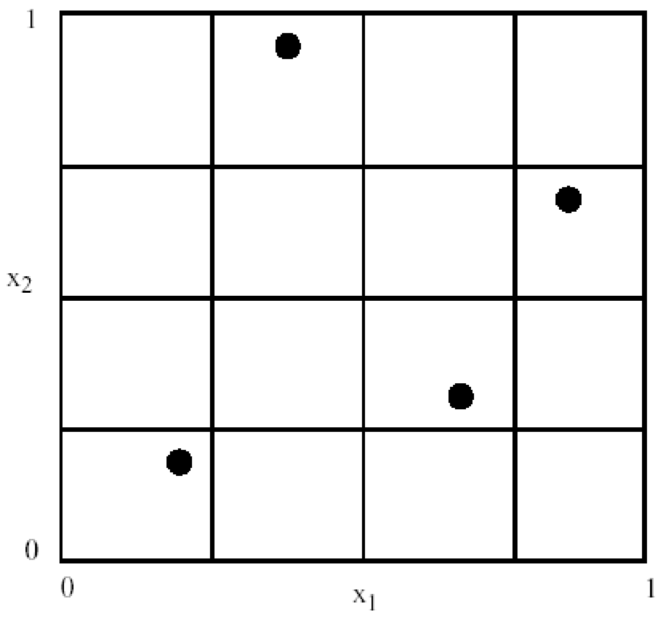


Figure 10.1: An example of Latin hypercube sampling with four bins in design parameters x_1 and x_2. The dots are the sample sites.

$$c_2 = x_2^2 - \frac{1}{2}x_1$$

will be used to demonstrate the application of sampling methods for uncertainty quantification where it is assumed that x_1 and x_2 are uniform uncertain variables on the interval $[0, 1]$. The DAKOTA input file for this problem is shown in Figure 10.2. The number of samples to perform is controlled with the `samples` specification, the type of sampling algorithm to use is controlled with the `sample_type` specification, the threshold values used for computing statistics on the response functions is specified with the `response_thresholds` input, and the seed specification controls the sequence of the pseudo-random numbers generated by the sampling algorithms. The input samples generated are shown in Figure 10.3 for the case where `samples` = 5 and `samples` = 10 for both `random` ($\circ$) and `lhs` ($+$) sample types.

Latin hypercube sampling ensures full coverage of the range of the input variables, which is often a problem with Monte Carlo sampling when the number of samples is small. In the case of `samples = 5`, poor stratification is evident in x_1 as four out of the five Monte Carlo samples are clustered in the range $0.35 < x_1 < 0.55$, and the regions $x_1 < 0.3$ and $0.6 < x_1 < 0.9$ are completely missed. For the case where `samples = 10`, some clustering in the Monte Carlo samples is again evident with 4 samples in the range $0.5 < x_1 < 0.55$. In both cases, the stratification with LHS is superior. The response function statistics returned by DAKOTA are shown in Figure 10.4. The first two blocks of output specify the mean responses, the standard deviations, and confidence intervals for the means of the response functions. The last section of the output specifies probability levels (points along a CDF or CCDF) for a response function at various response levels. Note that DAKOTA now allows the user to specify which format they would like to see the output, CDF or CCDF. Also, the user can specify response levels and get corresponding probabilities as output or specify various probability levels and get response levels as output. In this example, `distribution cumulative` was specified, and `response_levels` were set to obtain probabilities.

In addition to obtaining statistical summary information of the type shown in Figure 10.4, the results of LHS sampling now include correlations. Four types of correlations are returned in the output: simple and partial

```
    method,                                                         \
            nond_sampling,                                          \
              samples = 10 seed = 98765                             \
              response_levels = 0.1 0.2 0.6                         \
              0.1 0.2 0.6                                           \
              0.1 0.2 0.6                                           \
    #         sample_type random                                    \
              sample_type lhs                                       \
              distribution cumulative

    variables,                                                      \
    # Two uncertain uniform random variables on the interval [0,1]  \
            uniform_uncertain = 2                                   \
              uuv_dist_lower_bounds =  0  0                         \
              uuv_dist_upper_bounds =  1  1                         \
              uuv_descriptor        =  'x1'   'x2'

    interface,                                                      \
            application system asynch evaluation_concurrency = 5    \
            analysis_driver = 'text_book'

    responses,                                                      \
            num_response_functions = 3                              \
            no_gradients                                            \
            no_hessians
```

Figure 10.2: DAKOTA input file for UQ example using LHS sampling.

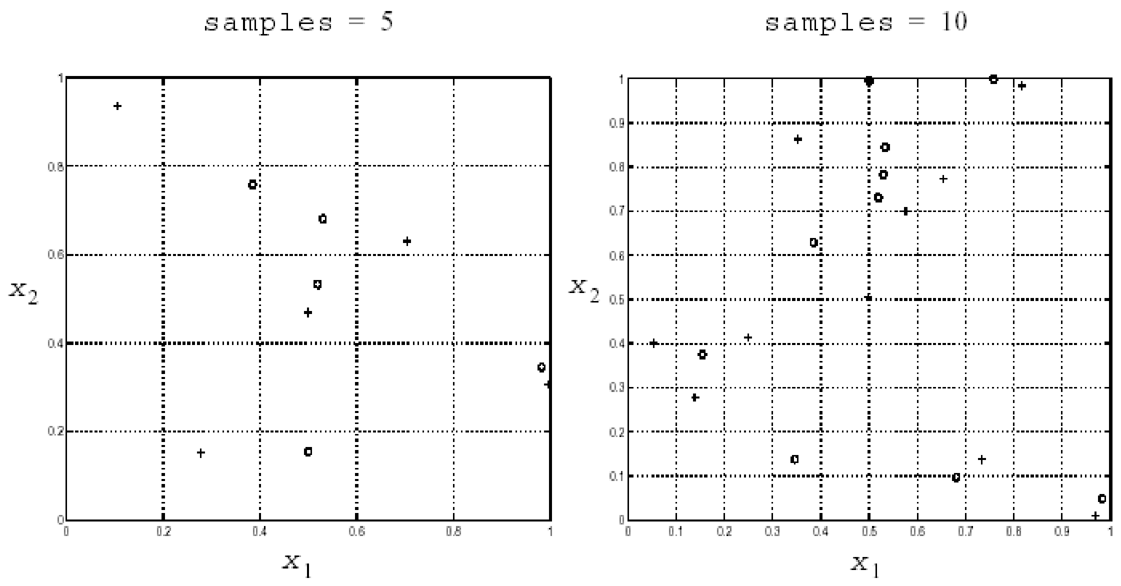


Figure 10.3: Distribution of input sample points for random (○) and lhs (+) sampling for `samples=5` and `10`.

```
Statistics based on 10 observations:

Moments for each response function:
response_fn1:  Mean = 3.83840e-01  Std. Dev. = 4.02815e-01
Coeff. of Variation = 1.04944e+00
response_fn2:  Mean = 7.47987e-02  Std. Dev. = 3.46861e-01
Coeff. of Variation = 4.63726e+00
response_fn3:  Mean = 7.09462e-02  Std. Dev. = 3.41532e-01
Coeff. of Variation = 4.81397e+00

95% confidence intervals for each response function:
response_fn1:  Mean = ( 1.34172e-01, 6.33507e-01 )
response_fn2:  Mean = ( -1.40188e-01, 2.89785e-01 )
response_fn3:  Mean = ( -1.40738e-01, 2.82630e-01 )

Probabilities for each response function:
Cumulative Distribution Function (CDF) for response_fn1:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   1.0000000000e-01   3.0000000000e-01
   2.0000000000e-01   5.0000000000e-01
   6.0000000000e-01   7.0000000000e-01
Cumulative Distribution Function (CDF) for response_fn2:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   1.0000000000e-01   5.0000000000e-01
   2.0000000000e-01   7.0000000000e-01
   6.0000000000e-01   9.0000000000e-01
Cumulative Distribution Function (CDF) for response_fn3:
     Response Level  Probability Level  Reliability Index
     --------------  -----------------  -----------------
   1.0000000000e-01   6.0000000000e-01
   2.0000000000e-01   6.0000000000e-01
   6.0000000000e-01   9.0000000000e-01
```

Figure 10.4: DAKOTA response function statistics from UQ sampling example.

```
Simple Correlation Matrix between input and output:
                       x1           x2 response_fn1 response_fn2 response_fn3
          x1  1.00000e-00
          x2 -7.22482e-02  1.00000e+00
response_fn1 -7.04965e-01 -6.27351e-01  1.00000e+00
response_fn2  8.61628e-01 -5.31298e-01 -2.60486e-01  1.00000e+00
response_fn3 -5.83075e-01  8.33989e-01 -1.23374e-01 -8.92771e-01  1.00000e+00

Partial Correlation Matrix between input and output:
             response_fn1 response_fn2 response_fn3
          x1 -9.65994e-01  9.74285e-01 -9.49997e-01
          x2 -9.58854e-01 -9.26578e-01  9.77252e-01

Simple Rank Correlation Matrix between input and output:
                       x1           x2 response_fn1 response_fn2 response_fn3
          x1  1.00000e+00
          x2 -6.66667e-02  1.00000e+00
response_fn1 -6.60606e-01 -5.27273e-01  1.00000e+00
response_fn2  8.18182e-01 -6.00000e-01 -2.36364e-01  1.00000e+00
response_fn3 -6.24242e-01  7.93939e-01 -5.45455e-02 -9.27273e-01  1.00000e+00

Partial Rank Correlation Matrix between input and output:
             response_fn1 response_fn2 response_fn3
          x1 -8.20657e-01  9.74896e-01 -9.41760e-01
          x2 -7.62704e-01 -9.50799e-01  9.65145e-01
```

Figure 10.5: Correlation results using LHS Sampling.

“raw” correlations, and simple and partial “rank” correlations. The raw correlations refer to correlations performed on the actual input and output data. Rank correlations refer to correlations performed on the ranks of the data. Ranks are obtained by replacing the actual data by the ranked values, which are obtained by ordering the data in ascending order. For example, the smallest value in a set of input samples would be given a rank 1, the next smallest value a rank 2, etc. Rank correlations are useful when some of the inputs and outputs differ greatly in magnitude: then it is easier to compare if the smallest ranked input sample is correlated with the smallest ranked output, for example.

Correlations are always calculated between two sets of sample data. One can calculate correlation coefficients between two input variables, between an input and an output variable (probably the most useful), or between two output variables. The simple correlation coefficients presented in the output tables are Pearson’s correlation coefficient, which is defined for two variables x and y as: $\texttt{Corr}(x, y) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$. Partial correlation coefficients are similar to simple correlations, but a partial correlation coefficient between two variables measures their correlation while adjusting for the effects of the other variables. For example, say one has a problem with two inputs and one output; and the two inputs are highly correlated. Then the correlation of the second input and the output may be very low after accounting for the effect of the first input. The rank correlations in DAKOTA are obtained using Spearman’s rank correlation. Spearman’s rank is the same as the Pearson correlation coefficient except that it is calculated on the rank data.

Figure 10.5 shows an example of the correlation output provided in DAKOTA. This example is the output from the input file in Figure 10.2 Note that these correlations are presently only available when one specifies lhs as the sampling method under nond_sampling. Also note that the simple and partial correlations should be similar in most cases (in terms of values of correlation coefficients). This is because we use a default “restricted pairing” method in the LHS routine which forces near-zero correlation amongst the inputs.

Finally, note that the LHS package can be used in design of experiments mode by including the `all_variables` flag in the method specification section of the DAKOTA input file. Then, instead of iterating on only the uncertain variables, the LHS package will sample on all of the continuous variables, where continuous design and continuous state variables are treated as having uniform probability distributions within their upper and lower bounds and any uncertain variables are sampled within their specified probability distributions.

10.3 Analytical Reliability Methods

Analytical reliability methods provide an alternative approach to uncertainty quantification which can be less computationally demanding than sampling techniques. Reliability methods for uncertainty quantification are based on probabilistic approaches that compute approximate response function distribution statistics based on specified uncertain variable distributions. These response statistics include response mean, response standard deviation, and cumulative or complementary cumulative distribution functions (CDF/CCDF). These methods are often more efficient at computing statistics in the tails of the response distributions (events with low probability) than sampling based approaches since the number of samples required to resolve a low probability can be prohibitive.

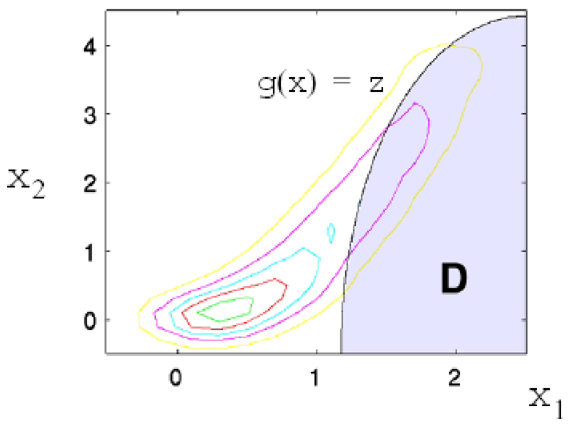
The methods all answer the fundamental question: “Given a set of uncertain input variables, $\mathbf{X}$, and a scalar response function, g, what is the probability that the response function is below a certain level, z?” Formally this can be written as $P[g(\mathbf{X}) < z] = F_g(z)$ where $F_g(z)$ is the cumulative distribution function (CDF) of the uncertain response $g(\mathbf{X})$ over a set of response levels.

This probability calculation involves a multi-dimensional integral over an irregularly shaped domain of interest, $\mathbf{D}$, where $g(\mathbf{X}) < z$ as displayed in Figure 10.6 for the case of two variables. The reliability methods all involve the transformation of the user-specified uncertain variables, $\mathbf{X}$, with probability density function, $p(x_1, x_2)$, which can be non-normal and correlated, to a space of independent Gaussian random variables, $\mathbf{u}$, possessing a mean value of zero and unit variance (i.e., standard normal variables). The region of interest, $\mathbf{D}$, is also mapped to the transformed space to yield, $\mathbf{D_u}$, where $g(\mathbf{U}) < z$ as shown in Figure 10.7. The Nataf transformation [12], which is identical to the Rosenblatt transformation [54] in the case of independent random variables, is used in DAKOTA to accomplish this mapping. This transformation is performed to make the probability calculation more tractable. In the transformed space, probability contours are circular in nature as shown in Figure 10.7 unlike in the original uncertain variable space, Figure 10.6. Also, the multi-dimensional integrals can be approximated by simple functions of a single parameter, β, called the reliability index. β is the minimum Euclidean distance from the origin in the transformed space to the response surface. This point is also known as the most probable point (MPP) of failure. Note, however, the methodology is equally applicable for generic functions, not simply those corresponding to failure criteria; this nomenclature is due to the origin of these methods within the disciplines of structural safety and reliability.

The determination of the MPP can be posed as a constrained optimization problem, where the objective function to be minimized is the distance from the origin to a surface in the unit-normal space. This surface defines an equality constraint for the minimization problem and the exact form of the constraint depends on the particular reliability method in use. The mean-value method (MV), advanced mean-value methods (AMV/AMV+) [58], and first order reliability method (FORM) are implemented in DAKOTA. These methods are explained below. A more thorough discussion of the methods can be found in the recent text by Haldar and Mahadevan [41].

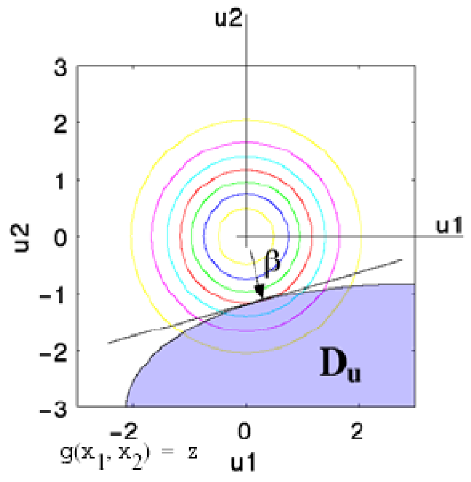
The Mean Value method (MV) is the simplest, least expensive reliability method in that it estimates the response means, response standard deviations, and the CDFs/CCDFs from a single evaluation of the response functions and gradients at the uncertain variable means. This approximation can have acceptable accuracy when the response functions are nearly linear and their distributions are approximately Gaussian, but can have very poor accuracy in other situations.

The expression for the approximate response mean μ_g, approximate response standard deviation σ_g , re-



$$P[g(\mathbf{X}) < z] = \iint\limits_{\mathbf{x} \in D} p(x_1, x_2) d\mathbf{x} = P[(\mathbf{x} \in \mathbf{D})]$$

Figure 10.6: Graphical depiction of calculation of cumulative distribution function in the original uncertain variable space.



$$P(\mathbf{X} \in D) = P(U \in D_U) \approx f(\beta)$$

Figure 10.7: Graphical depiction of integration for the calculation of cumulative distribution function in the transformed uncertain variable space.

sponse target to approximate probability/reliability level mapping $\bar{z} \rightarrow (p, \beta)$, and probability/reliability target to approximate response level mapping $\bar{p}, \bar{\beta} \rightarrow z$, are:

$$\mu_g = g(\mu_x)$$

$$\sigma_g = \sum_i \sum_j \mathtt{Cov}(i,j) \frac{d}{dx_i} g(\mu_x) \frac{d}{dx_j} g(\mu_x)$$

$$\beta_{CDF} = \frac{\mu_g - \bar{z}}{\sigma_g}, \beta_{CDF} = \frac{\bar{z} - \mu_g}{\sigma_g}$$

$$z = \mu_g - \sigma_g \bar{\beta}_{CDF}, z = \mu_g + \sigma_g \bar{\beta}_{CCDF}$$

where $\mathbf{x}$ are the uncertain values in the space of the original uncertain variables ("x-space"), and $g(\mathbf{x})$ is the limit state function is the limit state function (the response function for which probability-response level pairs are needed). The CDF reliability index β_{CDF}, the CCDF reliability index β_{CCDF}, the CDF probability $p(g \leq z)$, and the CCDF probability $p(g > z)$ are related to one another through:

$$p(g \leq z) = \Phi(-\beta_{CDF})$$

$$p(g > z) = \Phi(-\beta_{CCDF})$$

$$\beta_{CDF} = -\Phi^{-1}(p(g \leq z))$$

$$\beta_{CCDF} = -\Phi^{-1}(p(g > z))$$

$$\beta_{CDF} = -\beta_{CCDF}$$

All reliability methods except the MV method described above solve a nonlinear optimization problem to compute a most probable point (MPP) and then integrate about this point (rather than the uncertain variable means as in MV) to compute probabilities. The MPP search is performed in transformed standard normal space ("u-space") since it simplifies the probability integration: the distance of the MPP from the origin has the meaning of the number of standard deviations separating the mean response from a particular response threshold. The forward reliability analysis algorithm of computing CDF/CCDF probabilities for specified response levels is called the reliability index approach (RIA), and the inverse reliability analysis algorithm of computing response levels for specified CDF/CCDF probability levels is called the performance measure approach (PMA). The differences between the RIA and PMA formulations appear in the objective function and equality constraint formulations used in the MPP searches. For RIA, the MPP search for achieving the specified response level $\bar{z}$ is formulated as:

$$\text{minimize } \mathbf{u}^T \mathbf{u} \text{ subject to } G(\mathbf{u}) = \bar{z}$$

and for PMA, the MPP search for achieving the specified reliability/probability level beta, p is formulated as

$$\text{minimize } G(\mathbf{u}) \text{ subject to } \mathbf{u}^T\mathbf{u} = \bar{\beta}^2$$

where $\mathbf{u}$ is a vector centered at the origin in u-space, $g(\mathbf{x}) == G(\mathbf{u})$ by definition, and defines the CDF/CCDF probabilities. The sign of β is defined by:

$$G(u^* > G(0); \beta_{CDF} < 0, \beta_{CCDF} > 0$$

$$G(u^* < G(0); \beta_{CDF} > 0, \beta_{CCDF} < 0$$

where $G(O)$ is the median limit state response computed at the origin in u-space (where $p(g \leq z) = p(g > z) = 0.5$ and $\beta_{CDF} = \beta_{CCDF} = 0$.

There are a variety of algorithmic variations that can be explored within RIA/PMA reliability analysis. First, one may select among several different linearization approaches for the limit state function that can be used to reduce computational expense during the MPP searches. Options include:

1. A single linearization per response/probability level in x-space centered at the uncertain variable means (commonly known as the Advanced Mean Value (AMV) method).

$$g(x) \cong g(\mu_x) + \nabla_x g(\mu_x)^T (x - \mu_x)$$

2. The u-space AMV method. This is the same as AMV, except that the linearization is performed in u-space. This option has been termed the u-space AMV method.

$$G(u) \cong g(\mu_u) + \nabla_u g(\mu_u)^T (u - \mu_u)$$

3. AMV+. This method involves an initial x-space linearization at the uncertain variable means, with iterative relinearizations at each MPP estimate x* until the MPP converges (commonly known as the AMV+ method).

$$g(x) \cong g(x^*) + \nabla_x g(x^*)^T (x - x^*)$$

4. This is the same as AMV+, except that the linearizations are performed in u-space. This option has been termed the u-space AMV+ method.

$$G(u) \cong g(u^*) + \nabla_u g(u^*)^T (u - u^*)$$

5. Perform the MPP search on the original response functions without the use of any linearizations. The MPP search may be done with NPSOL or OPT++.

Thus, in summary, the user can choose to perform an RIA or PMA approach when implementing a reliability analysis. With either approach, there are a variety of methods to choose from to perform the linearization during MPP search: MV method, AMV, u-space AMV, AMV +, u-space AMV+, or direct optimization search for the MPP. Currently, the outputs for the MV technique consist of estimates of the mean and standard deviation of the response functions along with importance factors for each of the uncertain variables in the case of independent random variables. Each of the other methodologies (AMV, AMV+, FORM) output approximate values of the cumulative distribution function at the user-defined response levels.

(should I put a table in here with the various MPP search specification keywords?)

An example DAKOTA input file showing RIA using u-space AMV (number 2 above) is listed in Figure 10.8. This example quantifies the uncertainty in the response function

$$g(x_1, x_2) = \frac{x_1}{x_2} \tag{10.1}$$

by computing approximate response statistics using the u-space AMV method to determine the response cumulative distribution function

$$P([g(x_1, x_2)] < z) \tag{10.2}$$

X_1 and X_2 are independent, identically distributed lognormal random variables with means of `1` and standard deviations of `0.5`.

The formulation of a reliability analysis requires the user to specify the `nond_reliability` method. Then, the user specifies one possible search method for finding the MPP. In this example, we use `mpp_search u_linearize_mean` to specify that we are doing the AMV method in u-space. Finally, the user specifies response levels or probability/reliability levels to determine if the problem will be solved using an RIA approach or a PMA approach. In the example figure of 10.8, we use RIA, we use `response_levels` to specify the range of response levels for the problem. The resulting output for this input is shown in Figure 10.9, with probability levels and reliability levels listed for each response level (is there a good explanation for the differences in these terms?)

If the user specifies `nond_reliability` as a method with no additional specification on how to do the MPP search, then no MPP search is done: the Mean Value method is used. The MV results are shown in Figure 10.10 and consist of approximate mean and standard deviation of the response along with the importance factors for each uncertain variable. The importance factors are a measure of the sensitivity of the response function(s) to the uncertain input variables. The importance factors can be viewed as an extension of linear sensitivity analysis combining deterministic gradient information with input uncertainty information, *i.e.* input variable standard deviations. The accuracy of the importance factors is contingent of the validity of the linear approximation used to approximate the true response functions.

Should I leave in the last Figure 10.11 with the FORM comparison? Also, I need a better explanation how FORM relates to the `nond_reliability` case with the `no_linearize` case and nip or sqp.

10.4 Polynomial Chaos Methods

The objective of these techniques is to characterize the response of systems whose governing equations involve stochastic coefficients. The development of these techniques mirrors that of deterministic finite element analysis through the utilization of the concepts of projection, orthogonality, and weak convergence. The polynomial chaos expansion is based on a multidimensional Hermite approximation in standard normal random variables.

The coefficients for the terms in the polynomial chaos expansion are determined either from a coupled set of equations solved externally from the analysis package or from a set of statistical estimators known to converge to the Fourier coefficients, albeit at a rate that is unknown a priori. In DAKOTA, the latter approach is implemented where both direct Monte Carlo sampling and Latin hypercube sampling are available to serve as the estimators of the Fourier coefficients. A distinguishing feature of the methodology is that the solution series expansions are expressed as random processes, and not merely as statistics as is the case for many nondeterministic methodologies. This makes the technique particularly attractive for use in multi-physics applications which link different analysis packages. A more detailed explanation of the procedure can be found in Ghanem, et al. [29], [30].

```
interface,                                                    \
        application system asynch                             \
        analysis_driver = 'uq_example'

variables,                                                    \
        lognormal_uncertain = 2                               \
          lnuv_means             =  1.  1                     \
          lnuv_std_deviations    =  0.5 0.5                   \
          lnuv_descriptor        =  'TF1ln'   'TF2ln'         \
        uncertain_correlation_matrix =  1   0.3               \
                                        0.3 1

responses,                                                    \
        num_response_functions = 1                            \
#       analytic_gradients                                    \
        numerical_gradients                                   \
          method_source dakota                                \
          interval_type central                               \
          fd_step_size = 1.e-4                                \
        no_hessians

strategy,                                                     \
        single_method #graphics

method,                                                       \
        nond_reliability                                      \
          mpp_search u_linearize_mean                         \
          response_levels = .4 .5 .55 .6 .65 .7               \
           .75 .8 .85 .9 1. 1.05 1.15 1.2 1.25 1.30           \
           1.35 1.4 1.5 1.55 1.6 1.65 1.7 1.75
```

Figure 10.8: DAKOTA input file for UQ example using analytic reliability methods using an u-space AMV method.

```
    Cumulative Distribution Function (CDF) for response_fn1:
       Response Level  Probability Level  Reliability Index
       --------------  -----------------  -----------------
     5.4881163610e-01   1.3731709525e-01   1.0924526533e+00
     6.0653065972e-01   1.8131180592e-01   9.1037721105e-01
     6.3762815160e-01   2.0629637395e-01   8.1933948995e-01
     6.7032004601e-01   2.3321443990e-01   7.2830176884e-01
     7.0468808973e-01   2.6197643251e-01   6.3726404774e-01
     7.4081822071e-01   2.9245518576e-01   5.4622632663e-01
     7.7880078304e-01   3.2448677862e-01   4.5518860553e-01
     8.1873075308e-01   3.5787267059e-01   3.6415088442e-01
     8.6070797643e-01   3.9238311420e-01   2.7311316332e-01
     9.0483741803e-01   4.2776176043e-01   1.8207544221e-01
     1.0000000000e+00   5.0000000000e-01   3.0847722160e-17
     1.0512710964e+00   5.3626869081e-01  -9.1037721105e-02
     1.1618342427e+00   6.0761688580e-01  -2.7311316332e-01
     1.2214027582e+00   6.4212732941e-01  -3.6415088442e-01
     1.2840254167e+00   6.7551322138e-01  -4.5518860553e-01
     1.3498588075e+00   7.0754481424e-01  -5.4622632663e-01
     1.4190675486e+00   7.3802356749e-01  -6.3726404774e-01
     1.4918246977e+00   7.6678556010e-01  -7.2830176884e-01
     1.6487212707e+00   8.1868819408e-01  -9.1037721105e-01
     1.7332530179e+00   8.4168687601e-01  -1.0014149322e+00
     1.8221188004e+00   8.6268290475e-01  -1.0924526533e+00
     1.9155408289e+00   8.8169257045e-01  -1.1834903744e+00
     2.0137527074e+00   8.9876183893e-01  -1.2745280955e+00
     2.1170000166e+00   9.1396235904e-01  -1.3655658166e+00
```

Figure 10.9: Output from Analytical Reliability UQ example using FORM.

```
    -------------------------------------------------------------------
    MV Statistics for response_fn1:
      Approximate Mean Response                  = 1.0000000000e+00
      Approximate Standard Deviation of Response = 7.0710678119e-01
      Importance Factor for variable x1          = 5.0000000000e-01
      Importance Factor for variable x2          = 5.0000000000e-01
    Importance Factors are an extension of LINEAR sensitivity analysis.
    -------------------------------------------------------------------
```

Figure 10.10: Output from Analytical Reliability UQ example using MV.

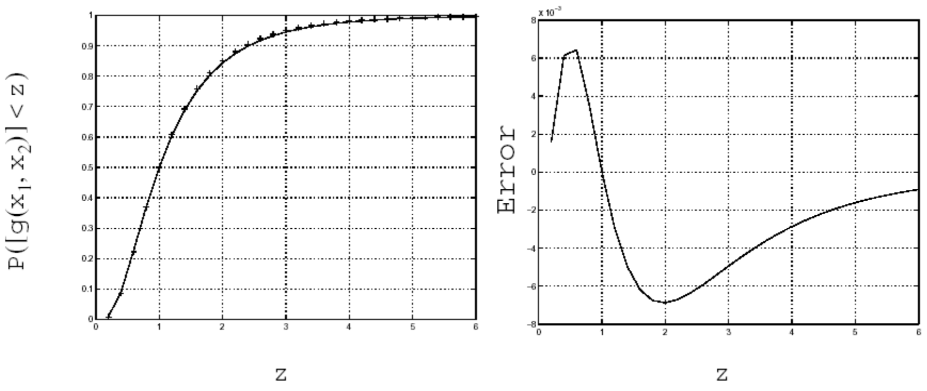


Figure 10.11: Comparison of the cumulative distribution function (CDF) computed by FORM (+ marks) and the exact CDF for $g(x_1, x_2) = \frac{x_1}{x_2}$

10.4.1 Uncertainty Quantification Example using Polynomial Chaos

A typical DAKOTA input file for performing an uncertainty quantification using polynomial chaos expansions is shown in Figure 10.12. The analysis involves the use of a `layered` model (defined in the '`UQ`' method specification) in order to manage the construction of a Hermite polynomial approximation (defined in the '`PCE`' interface specification) built using 250 LHS samples of the truth model uq_example (defined in the 'DACE' method and '`I1`' interface specifications).

After the Hermite polynomial surrogate model has been constructed, the `nond_polynomial_chaos` method performs a UQ analysis using 1000 LHS samples on the surrogate to compute estimates of the mean, standard deviation, coefficient of variation, and 95% confidence interval for the response function and the probability of exceeding the `response_thresholds` value. As shown in Figure 10.13, the method outputs these quantities in addition to the approximate coefficients in the polynomial chaos expansion for the response function. It should be noted that only standard normal random variables are supported in `nond_polynomial_chaos` at this time.

10.5 Future Nondeterministic Methods

Uncertainty analysis methods under investigation for future inclusion into the DAKOTA framework include extensions to the analytical reliability techniques and sampling capabilities supported. Advanced "smart sampling" techniques such as bootstrap sampling (BS), importance sampling (IS), quasi-Monte Carlo simulation (qMC), and Markov chain Monte Carlo simulation (McMC) are being investigated. Efforts have been initiated to allow for the possibility of non-traditional representations of uncertainty. These include interval analysis, Dempster-Shafer theory of evidence, possibility theory, and combinations of these. Finally, the tractability and efficacy of the more intrusive variant of stochastic finite element/polynomial chaos expansion methods, previously mentioned, is being assessed for possible implementation in DAKOTA.

```
strategy,                                                \
        single_method #graphics                          \
          method_pointer = 'UQ'

method,                                                  \
        id_method = 'UQ'                                 \
        model_pointer = 'UQ_M'                           \
        nond_polynomial_chaos                            \
          expansion_order = 2                            \
          samples = 1000 seed = 12347                    \
          sample_type lhs                                \
          response_levels = 0.5

model,                                                   \
        id_model = 'UQ_M'                                \
        surrogate global                                 \
          dace_method_pointer = 'DACE'                   \
          hermite

variables,                                               \
          normal_uncertain    =  2                       \
          nuv_means           =  0  0                    \
          nuv_std_deviations  =  1  1                    \
          nuv_descriptor      =  'n1' 'n2'               \

responses,                                               \
        num_response_functions = 1                       \
        no_gradients                                     \
        no_hessians


method,                                                  \
        id_method = 'DACE'                               \
        model_pointer = 'DACE_M'                         \
        nond_sampling                                    \
          samples = 250 seed = 1158                      \
          sample_type lhs

model,                                                   \
        id_model = 'DACE_M'                              \
        single                                           \
          interface_pointer = 'I1'

interface,                                               \
        id_interface = 'I1'                              \
        system asynchronous evaluation_concurrency = 5  \
          analysis_driver = 'log_ratio'
```

Figure 10.12: DAKOTA input file for performing UQ using polynomial chaos expansions.

```
-------------------------------------------------------------------
Statistics based on 1000 observations:

Moments for each response function:
response_fn1:  Mean = -2.785e+00  Std. Dev. = 4.940e+00
               Coeff. of Variation = -1.774e+00

95\% confidence intervals for each response function:
response_fn1:  Mean = ( -3.091e+00, -2.479e+00 )

Probabilities for each response function:
response_fn1: 83.000% below and 17.000% above the threshold value of
              5.00000e-01

Polynomial Chaos coefficients vector output
response_fn1
1       -2.7767149288e+00
2       -3.7452282807e+00
3       -6.5491680438e-03
4       -1.6293722861e+00
5        9.2459408840e-01
6        1.3637964830e+00
-------------------------------------------------------------------
```

Figure 10.13: Output from UQ analysis using polynomial chaos expansions.

Chapter 11

Optimization Capabilities

11.1 Overview

DAKOTA's optimization capabilities include a variety of gradient-based and nongradient-based optimization methods. Numerous packages are available, some of which are commercial packages, some of which are developed internally to Sandia, and some of which are free software packages from the open source community. The downloaded version of DAKOTA excludes the commercially developed packages but includes CONMIN, OPT++, COLINY, and PICO. Interfaces to DOT and NPSOL are provided with DAKOTA, but to use either of these commercial optimizers, the user must obtain a software license and the source code for these packages separately. The commercial software can then be compiled into DAKOTA by following DAKOTA's installation procedures (see notes in `/Dakota/INSTALL`).

DAKOTA's input commands permit the user to specify two-sided nonlinear inequality constraints of the form $g_{L_i} \leq g_i(\mathbf{x}) \leq g_{U_i}$, as well as nonlinear equality constraints of the form $h_j(\mathbf{x}) = h_{t_j}$ (see also Section 1.4.1). Some optimizers (e.g., NPSOL, OPT++) can handle these constraint forms directly, whereas other optimizers (e.g., DOT, CONMIN) require DAKOTA to perform an internal conversion of all constraints to one-sided inequality constraints of the form $g_i(\mathbf{x}) \leq 0$. In the latter case, the two-sided inequality constraints are treated as $g_i(\mathbf{x}) - g_{U_i} \leq 0$ and $g_{L_i} - g_i(\mathbf{x}) \leq 0$ and the equality constraints are treated as $h_j(\mathbf{x}) - h_{t_j} \leq 0$ and $h_{t_j} - h_j(\mathbf{x}) \leq 0$. The situation is similar for linear constraints: NPSOL and OPT++ support them directly, whereas DOT and CONMIN do not. For linear inequalities of the form $a_{L_i} \leq \mathbf{a}_i^T\mathbf{x} \leq a_{U_i}$ and linear equalities of the form $\mathbf{a}_i^T\mathbf{x} = a_{t_j}$, the nonlinear constraint arrays in DOT and CONMIN are further augmented to include $\mathbf{a}_i^T\mathbf{x} - a_{U_i} \leq 0$ and $a_{L_i} - \mathbf{a}_i^T\mathbf{x} \leq 0$ in the inequality case and $\mathbf{a}_i^T\mathbf{x} - a_{t_j} \leq 0$ and $a_{t_j} - \mathbf{a}_i^T\mathbf{x} \leq 0$ in the equality case. Awareness of these constraint augmentation procedures can be important for understanding the diagnostic data returned from the DOT and CONMIN algorithms.

When gradient and Hessian information are used in the optimization, it is assumed that derivative components will be computed only with respect to the *continuous design variables*. The omission of discrete variables from gradient vectors and Hessian matrices is common among all DAKOTA optimization methods; however, inclusion of only the continuous design variables differs from parameter study methods (which assume derivatives with respect to all continuous variables) and from nondeterministic analysis methods (which assume derivatives with respect to the uncertain variables).

11.2 Optimization Software Packages

11.2.1 COLINY Library

The COLINY library [42] supersedes the SGOPT library and contains a variety of nongradient-based optimization algorithms. The suite of COLINY optimizers available in DAKOTA currently include the following:

- **Global Optimization Methods**
  - Several evolutionary algorithms, including genetic algorithms (`coliny_ea`)
  - DIRECT [56] (`coliny_direct`)
- **Local Optimization Methods**
  - Solis-Wets (`coliny_solis_wets`)
  - Pattern Search (`coliny_pattern_search`)
- **Interfaces to Third-Party Local Optimization Methods**
  - Asynchronous Parallel Pattern Search (APPS) [44] [1] (`coliny_apps`)
  - COBYLA2 (`coliny_cobyla`)

For expensive optimization problems, COLINY's global optimizers are best suited for identifying promising regions in the global design space. In multimodal design spaces, the combination of global identification (from COLINY) with efficient local convergence (from DOT, NPSOL, CONMIN, or OPT++) can be highly effective. None of the COLINY methods are gradient-based, which makes them appropriate for problems for which gradient information is unavailable or is of questionable accuracy due to numerical noise. The COLINY methods support bound constraints and nonlinear constraints, but not linear constraints. Note that the nonlinear constraints are satisfied using penalty function formulations [57]. Refer to Table 17.1 for additional method classification information.

An example specification for a simplex-based pattern search algorithm from COLINY is:

```
method,                                         \
      coliny_pattern_search                     \
        max_function_evaluations = 2000         \
        solution_accuracy = 1.0e-4              \
        initial_delta = 0.05                    \
        threshold_delta = 1.0e-8                \
        pattern_basis simplex                   \
        exploratory_moves best_all              \
        contraction_factor = 0.75
```

The DAKOTA Reference Manual [17] contains additional information on the COLINY options and settings.

11.2.2 Constrained Minimization (CONMIN) Library

The CONMIN library [65] contains two methods for gradient-based nonlinear optimization. For constrained optimization, the Method of Feasible Directions (DAKOTA's `conmin_mfd` method selection) is available, while for unconstrained optimization, the Fletcher-Reeves conjugate gradient method

[1] http://software.sandia.gov/appspack/

(DAKOTA's `conmin_frcg` method selection) is available. Both of these methods are most efficient at finding a local minimum in the vicinity of the starting point. The methods in CONMIN can be applied to global optimization problems, but there is no guarantee that they will find the globally optimal design point.

One observed drawback to CONMIN's Method of Feasible Directions is that it does a poor job handling equality constraints. This is the case even if the equality constraint is formulated as two inequality constraints. This problem is what motivates the modifications to MFD that are present in DOT's MMFD algorithm. For problems with equality constraints, it is better to use the OPT++ nonlinear interior point methods, NPSOL, or one of DOT's constrained optimization methods (see below).

An example specification for CONMIN's Method of Feasible Directions algorithm is:

```
method,                                 \
      conmin_mfd                        \
        convergence_tolerance = 1.0e-4  \
        max_iterations = 100            \
        output quiet
```

Refer to the DAKOTA Reference Manual [17] for more information on the settings that can be used with CONMIN methods.

11.2.3 Design Optimization Tools (DOT) Library

The DOT library [67] contains nonlinear programming optimizers, specifically the Broyden-Fletcher-Goldfarb-Shanno (DAKOTA's `dot_bfgs` method selection) and Fletcher-Reeves conjugate gradient (DAKOTA's `dot_frcg` method selection) methods for unconstrained optimization, and the modified method of feasible directions (DAKOTA's `dot_mmfd` method selection), sequential linear programming (DAKOTA's `dot_slp` method selection), and sequential quadratic programming (DAKOTA's `dot_sqp` method selection) methods for constrained optimization.

All DOT methods are local gradient-based optimizers which are best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the NPSOL, CONMIN, and OPT++ libraries.

Through the `optimization_type` specification, DOT can be used to solve either minimization or maximization problems. For all other libraries (i.e., CONMIN, NPSOL, OPT++, COLINY), it is up to the user to reformulate a maximization problem as a minimization problem by negating the objective function (i.e., maximize $f(x)$ is equivalent to minimize $-f(x)$). An example specification for DOT's BFGS quasi-Newton algorithm is:

```
method,                               \
      dot_bfgs                        \
        optimization_type maximize    \
        convergence_tolerance = 1.0e-4  \
        max_iterations = 100          \
        output quiet
```

See the DAKOTA Reference Manual [17] for additional detail on the DOT commands. More information on DOT can be obtained by contacting Vanderplaats Research and Development at http://www.vrand.com.

11.2.4 JEGA

The JEGA (John Eddy's Genetic Algorithms) library contains two global optimization methods. The first is a Multi-objective Genetic Algorithm (MOGA) which performs Pareto optimization. The second is a

Single-objective Genetic Algorithm (SOGA) which performs optimization on a single objective function. The JEGA library was written by John Eddy, currently a member of the technical staff at Sandia. These functions are accessed as (`moga`, and `soga`) within DAKOTA. DAKOTA provides access to the JEGA library through the JEGAOptimizer class. The DAKOTA Reference Manual [17] contains additional information on the JEGA options and settings. Section 11.3 discusses multiobjective optimization in more detail, and there are some additional MOGA examples in Chapter 20.

11.2.5 MOOCHO Library

The MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) library, formerly known as rSQP++, is a new addition to DAKOTA that is not yet publicly available. It provides both general-purpose sequential quadratic programming (SQP) algorithms for nested analysis and design (NAND) as well as reduced-space SQP algorithms for simultaneous analysis and design (SAND). Additional information on SAND is provided in Section 11.3.2. MOOCHO algorithm capabilities are available using the `reduced_sqp` method selection.

11.2.6 NPSOL Library

The NPSOL library [31] contains a sequential quadratic programming (SQP) implementation (DAKOTA's `npsol_sqp` method selection). SQP is a nonlinear programming approach for constrained minimization which solves a series of quadratic programming (QP) subproblems. It uses an augmented Lagrangian merit function and a BFGS approximation to the Hessian of the Lagrangian. It is an infeasible method in that constraints will be satisfied at the final solution, but not necessarily during the solution process.

NPSOL's gradient-based approach is best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the DOT, CONMIN, and OPT++ libraries.

An example of an NPSOL specification is:

```
method,                                    \
      npsol_sqp                            \
        convergence_tolerance = 1.0e-6     \
        max_iterations = 100               \
        output quiet
```

See the DAKOTA Reference Manual [17] for additional detail on the NPSOL commands. More information on NPSOL can be obtained by contacting Stanford Business Software at http://www.sbsi-sol-optimize.com.

The NPSOL library generates diagnostics in addition to those appearing in the DAKOTA output stream. These diagnostics are written to the default FORTRAN device 9 file (e.g., `ftn09` or `fort.9`, depending on the architecture) in the working directory.

11.2.7 OPT++ Library

The OPT++ library [50] contains primarily nonlinear programming optimizers for unconstrained, bound constrained, and nonlinearly constrained minimization: Polak-Ribiere conjugate gradient (DAKOTA's `optpp_cg` method selection), quasi-Newton (DAKOTA's `optpp_q_newton` method selection), finite difference Newton (DAKOTA's `optpp_fd_newton` method selection), and full Newton (DAKOTA's `optpp_newton` method selection). The library also contains the parallel direct search nongradient-based method [13] (specified as DAKOTA's `optpp_pds` method selection).

OPT++'s gradient-based optimizers are best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. OPT++'s PDS method does not use gradients and has some limited global identification abilities; it is best suited for problems for which gradient information is unavailable or is of questionable accuracy due to numerical noise. Some OPT++ methods are strictly unconstrained (`optpp_cg`) and some support bound constraints (`optpp_pds`), whereas the Newton-based methods (`optpp_q_newton`, `optpp_fd_newton`, and `optpp_newton`) all support general linear and nonlinear constraints (refer to Table 17.1). Other gradient-based optimizers include the DOT, CONMIN, and NPSOL libraries. For least squares methods based on OPT++, refer to Section 12.2.1.

An example specification for the OPT++ quasi-Newton algorithm is:

```
    method,                                         \
          optpp_q_newton                            \
            max_iterations = 50                     \
            convergence_tolerance = 1e-4            \
            output debug
```

See the DAKOTA Reference Manual [17] for additional detail on the OPT++ commands.

The OPT++ library generates diagnostics in addition to those appearing in the DAKOTA output stream. These diagnostics are written to the file `OPT_DEFAULT.out` in the working directory.

11.2.8 Parallel Integer Combinatorial Optimization (PICO)

DAKOTA employs the branch and bound capabilities of the PICO library for solving discrete and mixed continuous/discrete constrained nonlinear optimization problems. This capability is implemented in DAKOTA as a strategy and is discussed further in Section 13.5.

11.2.9 SGOPT

The SGOPT library has been deprecated, and all methods have been migrated to the COLINY library.

11.3 Additional Optimization Capabilities

DAKOTA provides several capabilities which extend the services provided by the optimization software packages described in Section 11.2. First, any of the optimization algorithms can be used for multiobjective optimization problems through the use of weighted sum techniques. Second, large-scale optimization algorithms (e.g., MOOCHO) can be used for simultaneous analysis and design through the use of a fully-intrusive interface to internal simulation residual vectors and Jacobian matrices. Finally, with any optimizer (or least squares solver described in Section 12.2), user-specified (or in some cases automatic) scaling may be applied to any of continuous design variables, functions (or least squares terms), and constraints.

11.3.1 Multiobjective Optimization

Multiobjective optimization means that there are two or more objective functions that you wish to optimize simultaneously. Often these are conflicting objectives, such as cost and performance. The answer to a multi-objective problem is usually not a single point. Rather, it is a set of points called the Pareto front. Each point on the Pareto front satisfies the Pareto optimality criterion, which is stated as follows: a feasible vector X^* is Pareto optimal if there exists no other feasible vector X which would improve some objective without causing a simultaneous worsening in at least one other objective. Thus, if a feasible point X' exists that CAN be improved on one or more objectives simultaneously, it is not Pareto optimal: it is said to be ”dominated” and the points along the Pareto front are said to be ”non-dominated.”

There are two ways to approach a multiobjective problem. The most common approach is to combine multiple objectives into one, then use an appropriate optimization technique on the single objective function. This approach is available in DAKOTA and is outlined below. The advantage of this approach is that one is only solving a single objection problem, and can use an optimization method that is especially suited for the particular problem class. The disadvantage of this approach is that a linear weighted sum objective will not find optimal solutions if the true Pareto front is nonconvex. Also, if one wants to understand the effects of changing weights, this method can become computationally expensive. Since each optimization of a single weighted objective will find only one point near or on the Pareto front, many optimizations need to be performed to get a good parametric understanding of the influence of the weights.

DAKOTA offers two options for multiobjective problems: one is to transform the multiobjective problem into a single objective one. This approach is explained below. The other is to use a multiobjective genetic algorithm (`moga`) to create a population of nondominated solutions. The `moga` method is explained in Chapters 2 and 20. Over time, the selection operators of a genetic algorithm act to efficiently select solutions along the Pareto front. Because a GA is inherently parallel, an entire population in a GA can represent the Pareto front. Thus, although GAs are computationally expensive when compared to gradient-based methods, the advantage is that one can obtain an entire Pareto set at the end of one genetic algorithm run, as compared with having to run the “weighted sum” single objective problem multiple times with different weights.

The selection of a multiobjective optimization problem is made through the specification of multiple objective functions in the responses keyword block (i.e., the `num_objective_functions` specification is greater than `1`). The weighting factors on these objective functions can be optionally specified using the `multi_objective_weights` keyword (the default is equal weightings). The composite objective function for this optimization problem, F, is formed using these weights as follows: $F = \sum_{k=1}^{R} w_k f_k$, where the f_k terms are the individual objective function values, the terms are the weights, and is the number of objective functions. The weighting factors stipulate the relative importance of the design concerns represented by the individual objective functions; the higher the weighting factor, the more dominant a particular objective function will be in the optimization process.

Figure 11.1 shows a DAKOTA input file for a multiobjective optimization problem based on the “textbook” test problem. This input file is named `dakota_multiobj1.in` in the `/Dakota/test` directory. In the standard textbook formulation, there is one objective function and two constraints. In the multiobjective textbook formulation, all three of these functions are treated as objective functions (`num_objective_functions = 3`), with weights given by the `multi_objective_weights` keyword. Note that it is not required that the weights sum to a value of one. The multiobjective optimization capability also allows any number of constraints, although none are included in this example.

Figure 11.2 shows an excerpt of the results for this multiobjective optimization problem. The data for function evaluation 9 show that the simulator is returning the values and gradients of the three objective functions and that this data is being combined by DAKOTA into the value and gradient of the composite objective function, as identified by the header “`Multiobjective transformation:`”. This combination of value and gradient data from the individual objective functions employs the user-specified weightings of `.7`, `.2`, and `.1`. Convergence to the optimum of the multiobjective problem is indicated in this case by the gradient of the composite objective function going to zero (no constraints are active).

By performing multiple optimizations for different sets of weights, a family of optimal solutions can be generated which define the trade-offs that result when managing competing design concerns. This set of solutions is referred to as the Pareto set. Section 13.4 describes a solution strategy used for directly generating the Pareto set in order to investigate the trade-offs in multiobjective optimization problems.

```
# test file with a specific test.  The  is used to designate lines


strategy,                                                       \
        single_method                                           \
        tabular_graphics_data

method,                                                         \
        npsol_sqp                                               \
          convergence_tolerance = 1.e-8

variables,                                                      \
        continuous_design = 2                                   \
          cdv_initial_point    0.9    1.1                       \
          cdv_upper_bounds     5.8    2.9                       \
          cdv_lower_bounds     0.5   -2.9                       \
          cdv_descriptor       'x1'   'x2'

interface,                                                      \
        system asynchronous                                     \
          analysis_driver=  'text_book'

responses,                                                      \
        num_objective_functions = 3                             \
        multi_objective_weights = .7 .2 .1                      \
        analytic_gradients                                      \
        no_hessians                             \
```

Figure 11.1: Example DAKOTA input file for multiobjective optimization.

```
    ------------------------------
    Begin Function Evaluation    9
    ------------------------------
    Parameters for function evaluation 9:
                          5.9388064484e-01 x1
                          7.4158741199e-01 x2

    (text_book /var/tmp/qaagjayaZ /var/tmp/raahjayaZ)

    Active response data for function evaluation 9:
    Active set vector = { 3 3 3 }
                           3.1662048104e-02 obj_fn1
                          -1.8099485679e-02 obj_fn2
                           2.5301156720e-01 obj_fn3
     [ -2.6792982174e-01 -6.9024137409e-02  ] obj_fn1 gradient
     [  1.1877612897e+00 -5.0000000000e-01  ] obj_fn2 gradient
     [ -5.0000000000e-01  1.4831748240e+00  ] obj_fn3 gradient


    Multiobjective transformation:
                          4.3844693257e-02 obj_fn
     [  1.3827220000e-06  5.8621370000e-07  ] obj_fn gradient

        7    1 1.0E+00    9  4.38446933E-02 1.5E-06    2 T TT

     Exit NPSOL - Optimal solution found.

     Final nonlinear objective value =   0.4384469E-01
```

Figure 11.2: DAKOTA results for the multiobjective optimization example.

11.3.2 Simultaneous Analysis and Design (SAND) Optimization

DAKOTA was originally developed as a “black box” optimization tool that employs non-intrusive interfaces with simulation codes. While this approach is useful for many engineering design applications, it can become prohibitively expensive when there is a large design space (i.e., $O(10^2 - 10^3)$ design parameters) and when the computational simulation is highly nonlinear. Current research and development is underway to add a simultaneous analysis and design (SAND) capability to DAKOTA. This “all at once approach” is considerably more intrusive to a simulation code than any current interfacing capability in DAKOTA. But in some large-scale applications, the SAND method may be the only viable alternative for optimization.

The basic idea behind SAND is to converge a nonlinear simulation code at the same time that the optimality conditions are being converged. This amounts to applying the nonlinear simulation residual equations as equality constraints in the optimization problem and then using an infeasible optimization method (e.g., sequential quadratic programming) which only satisfies these equality constraints in the limit (i.e., at the final optimal solution). This can result in a significant computational savings over black-box optimization approaches which require a nonlinear simulation to be fully-converged on every function evaluation.

To implement a SAND technique, modifications to the simulation package are necessary so that the optimization software may have access to the internal residual vector and state Jacobian matrix used by the simulation solver. The SAND techniques can then leverage the internal linear algebra of the simulation package as appropriate in performing the search direction calculations. A SAND-type optimization does make certain assumptions about the simulation package, such as there is access to the state Jacobian matrix (although matrix free methods can be interfaced as well), exact values are used in the state Jacobian, an implicit numerical solution scheme is used, there are no discontinuities in the system, and steady state solutions are to be obtained (although SAND transient solution capabilities are under development). Many single physics, PDE-based simulation codes fall in this category. SAND approaches can be applied to more complex simulation codes, such as multi-physics packages, but substantial modifications are often needed to make SAND feasible in these cases.

Details on SAND-type optimization approaches may be found in [4], [6]. Additional details on the SAND implementation in DAKOTA will appear in future releases of this Users Manual.

11.3.3 Optimization with User-specified or Automatic Scaling

Some optimization problems involving design variables, objective functions, or constraints on vastly different scales may be solved more efficiently if these quantities are adjusted to a common scale (typically on the order of unity). With any optimizer (or least squares solver described in Section 12.2), user-specified or automatic scaling may be applied to any of continuous design variables, nonlinear inequality and equality constraints, and linear inequality and equality constraints. User-specified scaling may be applied to objective functions or least squares terms. Discrete variable scaling is not supported.

Scaling is enabled on a per-method basis for optimizers and least squares minimizers by including the `scaling` keyword in the relevant `method` specification in the DAKOTA input deck. When scaling is enabled, variables, functions, gradients, Hessians, etc., are transformed such that the optimizer iterates in scaled variable space, whereas evaluations of the computational model as specified in the interface are performed on the original problem scale. Therefore using scaling does not require rewriting the interface to the simulation code.

Scaling factors are specified through the keywords listed in Table 11.1, and are ignored if the `scaling` keyword is omitted from the `method` specification. Each `*_scales` keyword specifies no, one, or a vector of scale values to be applied to the corresponding variables or responses. If a single value is specified using any of these keywords it will apply to each component of the relevant vector, e.g., `cdv_scales = 3.0` will apply a characteristic scaling value of `3.0` to each continuous design variable. Valid entries in `*_scales` vectors include positive characteristic values (user-specified scale factors), `1.0` to exempt a component from scaling, or `0.0` for automatic scaling, if available for that component. Negative scale

Table 11.1: Keywords for specifying scaling factors.

keyword	input spec section	default behavior
`cdv_scales`	`variables`	automatic
`objective_function_scales`	`responses`	off (automatic not allowed)
`least_squares_term_scales`	`responses`	off (automatic not allowed)
`nonlinear_inequality_scales`	`responses`	automatic
`nonlinear_equality_scales`	`responses`	automatic
`linear_inequality_scales`	`method`	automatic
`linear_equality_scales`	`method`	automatic

values are not currently permitted.

When scaling is enabled, the following progression will be used to determine the type of scaling used on each component of a variables or response vector:

1. When a strictly positive characteristic value is specified, the quantity will be scaled by it.
2. If a zero or no characteristic value is specified, automatic scaling will be attempted according to the following scheme:
   (a) two-sided bounds scaled into the interval $[0, 1]$;
   (b) one-sided bound or targets scaled by the absolute value of the characteristic value, moving the bound or target to -1 or +1.
   (c) no bounds or targets: no automatic scaling possible, therefore no scaling for this component

Automatic scaling is not available for objective functions or least squares terms since they do not have bound constraints. *Caution:* The scaling hierarchy is followed for all problem variables and constraints when the `scaling` keyword is specified, so one must note the default scaling behavior for each component and manually exempt components with a scale value of `1.0`, if necessary.

Scaling for linear constraints specified through `linear_inequality_scales` or `linear_equality_scales` is applied *after* any (user-specified or automatic) continuous variable scaling. For example, for scaling mapping unscaled continuous design variables x to scaled variables $\tilde{x}$:

$$\tilde{x}^j = \frac{x^j - x^j_O}{x^j_M},$$

we have the following matrix system for linear inequality constraints

$$\begin{gathered}
a_L \leq A_i x \leq a_U \\
a_L \leq A_i \left(\mathrm{diag}(x_M)\tilde{x} + x_O\right) \leq a_U \\
a_L - A_i x_O \leq A_i \mathrm{diag}(x_M)\tilde{x} \leq a_U - A_i x_O \\
\tilde{a}_L \leq \tilde{A}_i \tilde{x} \leq \tilde{a}_U,
\end{gathered}$$

and user-specified or automatically computed scaling multipliers are applied to this final transformed system, which accounts for continuous design variable scaling. When automatic scaling is in use for linear constraints they are linearly scaled by characteristic values only, but not affinely into the interval $[0, 1]$.

Figure 11.3 demonstrates the use of several scaling keywords for the textbook optimization problem. The continuous design variable `x1` is scaled by a characteristic value of `4.0`, whereas `x2` is scaled automatically into $[0, 1]$ based on its bounds. The objective function will be scaled by a factor of `50.0`, the first nonlinear constraint by a factor of `15.0`, and the second nonlinear constraint is not scaled.

```
strategy,                                                       \
        single_method

method,                                                         \
        dot_mmfd,                                               \
        max_iterations = 50,                                    \
        convergence_tolerance = 1e-4

variables,                                                      \
        continuous_design = 2                                   \
        cdv_initial_point    0.9    1.1                         \
        cdv_upper_bounds     5.8    2.9                         \
        cdv_lower_bounds     0.5   -2.9                         \
        cdv_scales           4.0    0.0                         \
        cdv_descriptor       'x1'   'x2'

interface,                                                      \
        fork                                                    \
          analysis_driver = 'text_book'                         \

responses,                                                      \
        num_objective_functions = 1                             \
        objective_function_scales 50.0                          \
        num_nonlinear_inequality_constraints = 2                \
        nonlinear_inequality_constraint_scales 15.0  1.0        \
        numerical_gradients                                     \
          method_source dakota                                  \
          interval_type central                                 \
          fd_gradient_step_size = 1.e-4                         \
        no_hessians
```

Figure 11.3: Sample usage of scaling keywords in DAKOTA input specification.

Chapter 12

Nonlinear Least Squares for Parameter Estimation

12.1 Overview

Nonlinear least squares methods are optimization algorithms which exploit the special structure of a sum of the squares objective function [32]. These problems commonly arise in parameter estimation, system identification, and test/analysis reconciliation. In order to exploit the problem structure, more granularity is needed in the response data than that required for a typical optimization problem. That is, rather than using the sum-of-squares objective function and its gradient, least squares iterators require each term used in the sum-of-squares formulation along with its gradient. This means that the m functions in the DAKOTA response data set consist of the individual least squares terms along with any nonlinear inequality and equality constraints. These individual terms are often called `residuals` in cases where they denote errors of observed quantities from desired quantities.

The enhanced granularity needed for nonlinear least-squares algorithms allows for simplified computation of an approximate Hessian matrix. These methods approximate the true Hessian matrix by neglecting terms in which the residual function values appear, under the assumption that the residuals tend towards zero at the solution. As a result, residual function value and gradient information is sufficient to define the value, gradient, and approximate Hessian of the sum-of-squares objective function. See Section 1.4.2 for additional details on this approximation.

In practice, least squares solvers will tend to be significantly more efficient than general-purpose optimization algorithms when the Hessian approximation is a good one, i.e., when the residuals tend towards zero at the solution. Specifically, they can exhibit the quadratic convergence rates of full Newton methods, even though only first-order information is used. Least squares solvers may experience difficulty when the residuals at the solution are significant.

In order to specify a least-squares problem, the responses section of the DAKOTA input should be configured using `num_least_squares_terms` (as opposed to `num_objective_functions` in the case of optimization). Any linear or nonlinear constraints are handled in an identical way to that of optimization (see Section 11.1; note that neither Gauss-Newton nor NLSSOL require any constraint augmentation). Gradients of the least squares terms and nonlinear constraints are required and should be specified using either `numerical_gradients`, `analytic_gradients`, or `mixed_gradients`. Since second derivatives of the least squares terms are not needed by nature of the Gauss-Newton approximation, the `no_hessians` specification should be used (exception: the derivative-order mismatch for nonlinearly-constrained Gauss-Newton described in Section 12.2.1 requires a specification of `analytic_hessians`). DAKOTA's scaling options, described in Section 11.3.3 can be used on least squares problems, using the `least_squares_term_scales` keyword to scale least squares residuals, if desired.

12.2 Solution Techniques

Nonlinear least squares problems can be solved using either the Gauss-Newton algorithm, which leverages the full Newton method from OPT++, the NLSSOL algorithm, which is closely related to NPSOL, or the NL2SOL algorithm which uses a secant-based algorithm. Details of each are provided below.

12.2.1 Gauss-Newton

DAKOTA's Gauss-Newton algorithm consists of combining an implementation of the Gauss-Newton Hessian approximation (see Section 1.4.2) with full Newton optimization algorithms from the OPT++ package [50]. This approach can be selected using the `optpp_g_newton` method specification. An example specification follows:

```
method,                                         \
      optpp_g_newton                            \
        max_iterations = 50                     \
        convergence_tolerance = 1e-4            \
        output debug
```

Refer to the DAKOTA Reference Manual [17] for more detail on the input commands for the Gauss-Newton algorithm.

The Gauss-Newton algorithm is gradient-based and is best suited for efficient navigation to a local least squares solution in the vicinity of the initial point. Global optima in multimodal design spaces may be missed. Gauss-Newton supports bound, linear, and nonlinear constraints. However, for the generally-constrained case, a derivative order mismatch exists in that the nonlinear interior point full-Newton algorithm will require second-order information for the nonlinear constraints whereas the Gauss-Newton approximation for the objective function Hessian only requires first order information for the least squares terms. This will be addressed in future releases through the use of quasi-Newton approximations to the constraint Hessians.

12.2.2 NLSSOL

The NLSSOL algorithm is a commercial software product of Stanford University that is bundled with current versions of the NPSOL library. It uses an SQP-based approach to solve generally-constrained nonlinear least squares problems. It periodically employs the Gauss-Newton Hessian approximation to accelerate the search. Its derivative order is balanced in that it requires only first-order information for the least squares terms and nonlinear constraints. This approach can be selected using the `nlssol_sqp` method specification. An example specification follows:

```
method,                                         \
      nlssol_sqp                                \
        convergence_tolerance = 1e-8
```

Refer to the DAKOTA Reference Manual [17] for more detail on the input commands for NLSSOL.

12.2.3 NL2SOL

The NL2SOL algorithm [REF] is a secant-based least-squares algorithm that is q-superlinearly convergent.

```
    Active response data for function evaluation 1:
    Active set vector = { 3 3 }
                          6.0000000000e-01 least_sq_term1
                          2.0000000000e-01 least_sq_term2
     [ -1.6000000000e+01  1.0000000000e+01  ] least_sq_term1 gradient
     [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient


        nlf2_evaluator_gn results: objective fn. =
        4.0000000000e-01
        nlf2_evaluator_gn results: objective fn. gradient =
     [ -1.9600000000e+01  1.2000000000e+01 ]
        nlf2_evaluator_gn results: objective fn. Hessian =
    [[  5.1400000000e+02 -3.2000000000e+02
       -3.2000000000e+02  2.0000000000e+02 ]]
```

Figure 12.1: Example of the Gauss-Newton approximation.

12.2.4 Future plans

The least squares branch in DAKOTA is an area of continuing enhancements, particularly through the addition of new least squares algorithms. One potential future addition is the orthogonal distance regression (ODR) algorithms which estimate values for both independent and dependent parameters.

12.3 Examples

Both the Rosenbrock and textbook example problems can be formulated as nonlinear least squares problems. Refer to Chapter 20 for more information on these formulations. Figure 12.1 shows an excerpt from the textbook example which demonstrates use of the Gauss-Newton approximation in computing the objective function value, gradient, and Hessian from values and gradients of the least squares terms.

Chapter 13

Advanced Optimization Strategies

13.1 Overview

DAKOTA's strategy capabilities were developed in order to provide a control layer for managing multiple iterators and models. It was driven by the observed need for "meta-optimization" and other high level systems analysis procedures in real-world engineering design problems. This capability allows the use of existing iterative algorithm and computational model software components as building blocks to accomplish more sophisticated studies, such as hybrid optimization, surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty.

13.2 Multilevel Hybrid Optimization

In the multilevel hybrid optimization strategy (keyword: `multi_level`), a sequence of optimization methods are applied to find an optimal design point. The goal of this strategy is to exploit the strengths of different optimization algorithms through different stages of the optimization process. Global/local hybrids (e.g., genetic algorithms combined with nonlinear programming) are a common example in which the desire for a global optimum is balanced with the need for efficient navigation to a local optimum. An important related feature is that the sequence of optimization algorithms can employ models of varying fidelity. In the global/local case, for example, it would often be advantageous to use a low-fidelity model in the global search phase, followed by use of a more refined model in the local search phase.

The specification for multilevel optimization involves a list of method identifier strings, and each of the corresponding method specifications has the responsibility for identifying the variables, interface, and responses specifications that each method will use (see the DAKOTA Reference Manual [17] and the example discussed below). Currently, only the uncoupled multilevel approach is available. The `coupled` and `uncoupled adaptive` approaches are placeholders for future capabilities.

In the uncoupled multilevel optimization approach, a sequence of optimization methods is invoked in the order specified in the DAKOTA input file. The best solution from each method is used as the starting point for the following method. Method switching is governed by the separate convergence controls of each method; that is, *each method is allowed to run to its own internal definition of completion without interference*. Individual method completion may be determined by convergence criteria (e.g., `convergence_tolerance`) or iteration limits (e.g., `max_iterations`).

Figure 13.1 shows a DAKOTA input file that specifies a multilevel optimization strategy to solve the "textbook" optimization test problem. This input file is named `dakota_multilevel.in` in the `/Dakota/test` directory. The three optimization methods are identified using the `method_list` specification in the strategy section of the input file. The identifier strings listed in the specification are

'GA' for genetic algorithm, 'PS' for pattern search, and 'NLP' for nonlinear programming. Following the strategy keyword block are three method keyword blocks. Note that each method has a tag following the id_method keyword that corresponds to one of the method names listed in the strategy keyword block. By following the keyword tags for the interface_pointer, variables_pointer, and responses_pointer, it is easy to see the specification linkages for this problem. The GA optimizer runs first and uses the variables keyword block 'V1', the interface keyword block 'I1', and the responses keyword block 'R1'. Once the GA is complete, the PS optimizer begins operation, and uses the best GA result as its starting point. The PS method again uses 'V1', 'I1', and 'R1'. Since both GA and PS are nongradient-based optimization methods, there is no need for gradient or Hessian information in the response keyword block. The NLP optimizer runs last, using the best result from the PS method as its starting point. It also uses the 'V1' and 'I1' keyword blocks, but it uses the responses keyword block 'R2' since the full Newton optimizer used in this example (optpp_newton) needs analytic gradient and Hessian data to perform its search.

13.3 Multistart Local Optimization

A simple, heuristic, global optimization technique is to use many local optimization runs, each of which is started from a different initial point in the parameter space. This is known as multistart local optimization. This is an attractive strategy in situations where multiple local optima are known or expected to exist in the parameter space. However, there is no theoretical guarantee that the global optimum will be found. This approach combines the efficiency of local optimization methods with a user-specified global stratification (using a specified starting_points list, a number of specified random_starts, or both; see the Reference Manual [17] for additional specification details). Since solutions for different starting points are independent, parallel computing may be used to concurrently run the local optimizations.

An example input file for multistart local optimization on the "quasi_sine" test function (see quasi_sine_fcn.C in /Dakota/test) is shown in Figure 13.2. The strategy keyword block in the input file contains the keyword multi_start, along with the set of starting points that will be used for the optimization runs. The other keyword blocks in the input file are similar to what would be used in a single optimization run.

The quasi_sine test function has multiple local minima, but there is an overall trend in the function that tends toward the global minimum at $(x1, x2) = (0.177, 0.177)$. See [35] for more information on this test function. Figure 13.3 shows the results summary for the five local optimizations performed. From the five starting points (as identified by the x1, x2 headers), the five local optima (as identified by the x1*, x2* headers) are all different and only one of the local optimizations finds the global minimum.

13.4 Pareto Optimization

The Pareto optimization strategy (keyword: pareto_set) is related to the multiobjective optimization capabilities discussed in Section 11.3.1. However, in a Pareto optimization strategy, multiple sets of multiobjective weightings will be evaluated. The user can specify these weighting sets in the strategy keyword block using a multi_objective_weight_sets list, a number of random_weight_sets, or both (see the Reference Manual [17] for additional specification details). Figure 13.4 shows the input commands from the file dakota_pareto.in in the /Dakota/test directory.

DAKOTA performs one multiobjective optimization problem for each set of multiobjective weights. The collection of computed optimal solutions form a Pareto set, which can be useful in making trade-off decisions in engineering design. Since solutions for different multiobjective weights are independent, parallel computing may be used to concurrently execute the multiobjective optimization problems.

Figure 13.5 shows the results summary for the Pareto-set optimization strategy. For the four multiobjective weighting sets (as identified by the w1, w2, w3 headers), the local optima (as identified by the x1,

```
strategy,                                       \
        graphics                                \
        multi_level uncoupled                   \
          method_list = 'GA' 'PS' 'NLP'

method,                                         \
        id_method = 'GA'                        \
        model_pointer = 'M1'                    \
        coliny_ea                               \
          seed = 1234                           \
          population_size = 10                  \
          verbose output

method,                                         \
        id_method = 'PS'                        \
        model_pointer = 'M1'                    \
        coliny_pattern search stochastic        \
          seed = 1234                           \
          initial_delta = 0.1                   \
          threshold_delta = 1.e-4               \
          solution_accuracy = 1.e-10            \
          exploratory_moves basic_pattern       \
          verbose output

method,                                         \
        id_method = 'NLP'                       \
        model_pointer = 'M2'                    \
        optpp_newton                            \
          gradient_tolerance = 1.e-12           \
          convergence_tolerance = 1.e-15        \
          verbose output

model,                                          \
        id_model = 'M1'                         \
        single                                  \
          variables_pointer = 'V1'              \
          interface_pointer = 'I1'              \
          responses_pointer = 'R1'              \

model,                                          \
        id_model = 'M2'                         \
        single                                  \
          variables_pointer = 'V1'              \
          interface_pointer = 'I1'              \
          responses_pointer = 'R2'              \

variables,                                      \
        id_variables = 'V1'                     \
        continuous_design = 2                   \
          cdv_initial_point    0.6    0.7       \
          cdv_upper_bounds     5.8    2.9       \
          cdv_lower_bounds     0.5   -2.9       \
          cdv_descriptor       'x1'   'x2'

interface,                                      \
        id_interface = 'I1'                     \
        direct                                  \
          analysis_driver=  'text_book'

responses,                                      \
        id_responses = 'R1'                     \
        num_objective_functions = 1             \
        no_gradients                            \
        no_hessians

responses,                                      \
        id_responses = 'R2'                     \
        num_objective_functions = 1             \
        analytic_gradients                      \
        analytic_hessians
```

Figure 13.1: DAKOTA input file for the multilevel optimization strategy.

```
strategy,                                      \
        multi_start graphics                   \
          method_pointer = 'NLP'               \
          random_starts = 3 seed = 123         \
          starting_points = -.8  -.8           \
                            -.8   .8           \
                             .8  -.8           \
                             .8   .8           \
                              0.   0.

method,                                        \
        id_method = 'NLP'                      \
        dot_bfgs

variables,                                     \
        continuous_design = 2                  \
          cdv_lower_bounds    -1.0     -1.0    \
          cdv_upper_bounds     1.0      1.0    \
          cdv_descriptor       'x1'     'x2'

interface,                                     \
        system #asynchronous                   \
          analysis_driver = 'quasi_sine_fcn'

responses,                                     \
        num_objective_functions = 1            \
        analytic_gradients                     \
        no_hessians
```

Figure 13.2: DAKOTA input file for the multistart local optimization strategy.

```
<<<<< Results summary:
   set_id            x1            x2           x1*           x2*        obj_fn
        1          -0.8          -0.8 -0.8543728665 -0.8543728665  0.5584096919
        2          -0.8           0.8 -0.9998398719   0.177092822   0.291406596
        3           0.8          -0.8   0.177092822 -0.9998398719   0.291406596
        4           0.8           0.8  0.1770928217  0.1770928217  0.0602471946
        5             0             0 0.03572926375 0.03572926375 0.08730499239
```

Figure 13.3: DAKOTA results summary for the multistart local optimization strategy.

```
strategy,                                                    \
        pareto_set graphics                                  \
          opt_method_pointer = 'NLP'                         \
          multi_objective_weight_sets =                      \
                                1.   0.   0.                 \
                                0.   1.   0.                 \
                                0.   0.   1.                 \
                              .333 .333 .333

method,                                                      \
        id_method = 'NLP'                                    \
        dot_bfgs

variables,                                                   \
        continuous_design = 2                                \
          cdv_initial_point     0.9    1.1                   \
          cdv_upper_bounds      5.8    2.9                   \
          cdv_lower_bounds      0.5   -2.9                   \
          cdv_descriptor        'x1'   'x2'

interface,                                                   \
        system #asynchronous                                 \
          analysis_driver = 'text_book'

responses,                                                   \
        num_objective_functions = 3                          \
        analytic_gradients                                   \
        no_hessians
```

Figure 13.4: DAKOTA input file for the Pareto optimization strategy.

`x2` headers) are all different and correspond to individual objective function values of $(f1, f2, f3) = (0.0, 0.5, 0.5), (13.1, -1.2, 8.16), (532., 33.6, -2.9)$, and $(0.125, 0.0, 0.0)$ (note: the composite objective function is tabulated under the `obj_fn` header). The first three solutions reflect exclusive optimization of each of the individual objective functions in turn, whereas the final solution reflects a balanced weighting and the lowest sum of the three objectives. Plotting these $(f1, f2, f3)$ triplets on a 3-dimensional plot results in a Pareto surface (not shown), which is useful for visualizing the trade-offs in the competing objectives.

```
<<<<< Results summary:
   set_id            w1            w2            w3            x1            x2        obj_fn
        1             1             0             0  0.9996554048   0.997046351 7.612301561e-11
        2             0             1             0           0.5           2.9          -1.2
        3             0             0             1           5.8 1.12747589e-11          -2.9
        4         0.333         0.333         0.333           0.5  0.5000000041      0.041625
```

Figure 13.5: DAKOTA results summary for the Pareto-set optimization strategy.

13.5 Mixed Integer Nonlinear Programming (MINLP)

Many nonlinear optimization problems involve a combination of discrete and continuous variables. These are known as mixed integer nonlinear programming (MINLP) problems. A typical MINLP optimization problem is formulated as follows:

$$\begin{aligned}
\text{minimize:} \quad & f(\mathbf{x}, \mathbf{d}) \\
\text{subject to:} \quad & \mathbf{g}_L \leq \mathbf{g}(\mathbf{x}, \mathbf{d}) \leq \mathbf{g}_U \\
& \mathbf{h}(\mathbf{x}, \mathbf{d}) = \mathbf{h}_t \\
& \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U \\
& \mathbf{d} \in \{-2, -1, 0, 1, 2\}
\end{aligned} \tag{13.1}$$

where $\mathbf{d}$ is a vector whose elements are integer values. In situations where the discrete variables can be temporarily relaxed (i.e., noncategorical discrete variables, see Section 4.2.2), the branch-and-bound algorithm can be applied. Categorical variables (e.g., true/false variables, or binary state variables) that are inherently discrete cannot be used with the branch and bound strategy. During the branch and bound process, the discrete variables are treated as continuous variables and the integrality conditions on these variables are incrementally enforced through a sequence of optimization subproblems. By the end of this process, an optimal solution that is feasible with respect to the integrality conditions is computed.

DAKOTA's branch and bound strategy (keyword: `branch_and_bound`) can solve optimization problems having either discrete or mixed continuous/discrete variables. This strategy uses the parallel branch-and-bound algorithm from the PICO software package [15], [16] to generate a series of optimization subproblems ("branches"). These subproblems are solved as continuous variable problems using any of DAKOTA's nonlinear optimization algorithms (e.g., DOT, NPSOL). When a solution to a branch is feasible with respect to the integrality constraints, it provides an upper bound on the optimal solution, which can be used to prune branches with higher objective functions which are not yet feasible. Since solutions for different branches are independent, parallel computing may be used to concurrently execute the optimization subproblems.

PICO, by itself, targets the solution of mixed integer linear programming (MILP) problems, and through coupling with DAKOTA's nonlinear optimizers, is extended to solution of MINLP problems. In the case of MILP problems, the upper bound obtained with a feasible solution is an exact bound and the branch and bound process is provably convergent to the global minimum. For nonlinear problems which may exhibit nonconvexity or multimodality, the process is heuristic in general, since there may be good solutions that are missed during the solution of a particular branch. However, the process still computes a series of locally optimal solutions, and is therefore a natural extension of the results from local optimization techniques for continuous domains. Only with rigorous global optimization of each branch can a global minimum be guaranteed when performing branch and bound on nonlinear problems of unknown structure.

In cases where there are only a few discrete variables and when the discrete values are drawn from a small set, then it may be reasonable to perform a separate optimization problem for all of the possible combinations of the discrete variables. However, this brute force approach becomes computationally intractable if these conditions are not met. The branch-and-bound algorithm will generally require solution of fewer subproblems than the brute force method, although it will still be significantly more expensive than solving a purely continuous design problem.

13.5.1 Example MINLP Problem

As an example, consider the following MINLP problem [20]:

$$
\begin{aligned}
\text{minimize:} \quad & f(\mathbf{x}) = \sum_{i=1}^{6}(x_i - 1.4)^4 \\
& g_1 = x_1^2 - \frac{x_2}{2} \le 0 \\
& g_2 = x_2^2 - \frac{x_1}{2} \le 0 \\
& -10 \le x_1, x_2, x_3, x_4 \le 10 \\
& x_5, x_6 \in \{0, 1, 2, 3, 4\}
\end{aligned} \tag{13.2}
$$

This problem is a variant of the textbook test problem described in Chapter 20. In addition to the introduction of two integer variables, a modified value of 1.4 is used inside the quartic sum to render the continuous solution a non-integral solution. Figure 13.6 shows a DAKOTA input file for solving this problem. This input file is named `dakota_bandb.in` in the `/Dakota/test` directory. Note the specification for the discrete variables, where lower and upper bounds are given. The discrete variables can take on any integer value within these bounds.

Figure 13.7 shows the sequence of branches generated for this problem. The first optimization subproblem relaxes the integrality constraint on parameters x_5 and x_6, so that $0 \le x_5 \le 4$ and $0 \le x_6 \le 4$. The values for x_5 and x_6 at the solution to this first subproblem are $x_5 = x_6 = 1.4$. Since x_5 and x_6 must be integers, the next step in the solution process "branches" on parameter x_5 to create two new optimization subproblems; one with $0 \le x_5 \le 1$ and the other with $2 \le x_5 \le 4$. Note that, at this first branching, the bounds on x_6 are still $0 \le x_6 \le 4$. Next, the two new optimization subproblems are solved. Since they are independent, they can be performed in parallel. The branch-and-bound process continues, operating on both x_5 and x_6 , until a optimization subproblem is solved where x_5 and x_6 are integer-valued. At the solution to this problem, the optimal values for x_5 and x_6 are $x_5 = x_6 = 1$.

In this example problem, the branch-and-bound algorithm executes as few as five and no more than seven optimization subproblems to reach the solution. For comparison, the brute force approach would require 25 optimization problems to be solved (i.e., five possible values for each of x_5 and x_6).

In the example given above, the discrete variables are integer-valued. In some cases, the discrete variables may be real-valued, such as $x \in \{0.0, 0.5, 1.0, 1.5, 2.0\}$. The branch-and-bound algorithm is restricted to work with integer values. Therefore, it is up to the user to perform a transformation between the discrete integer values from DAKOTA and the discrete real values that are passed to the simulation code (see Section 4.2.2). When integrality is not being relaxed, a common mapping is to use the integer value from DAKOTA as the index into a vector of discrete real values. However, when integrality is relaxed, additional logic for interpolating between the discrete real values is needed.

13.6 Optimization Under Uncertainty (OUU)

The nondeterministic optimization strategy (a.k.a. optimization under uncertainty) incorporates an uncertainty quantification method within the optimization process. This is often needed in engineering design problems when one must include the effect of input parameter uncertainties on the response functions of interest. A typical engineering example of OUU would minimize the probability of failure of a structure for a set of applied loads, where there is uncertainty in the loads and/or material properties of the structural components.

In the OUU strategy in DAKOTA, a nondeterministic method is used to evaluate the effect of uncertain variable distributions on response functions of interest (refer to Chapter 10 for additional information on nondeterministic analysis). Statistics on these response functions are then included in the objective and constraint functions of an optimization process. Three approaches are currently supported: nested OUU, surrogate-based OUU, and trust-region surrogate-based OUU. Additional details and computational results

```
strategy,                                                \
        branch_and_bound                                 \
          opt_method_pointer = 'NLP'                     \

method,                                                  \
        npsol_sqp                                        \
          id_method = 'NLP'                              \
          convergence_tol = 1.e-8

variables,                                               \
        continuous_design = 4                            \
          cdv_initial_point      0.5   1.5   0.5   1.5  \
          cdv_lower_bounds     -10.0 -10.0 -10.0 -10.0  \
          cdv_upper_bounds      10.0  10.0  10.0  10.0  \
        discrete_design = 2                              \
          ddv_initial_point      2     2                 \
          ddv_lower_bounds       0     0                 \
          ddv_upper_bounds       4     4

interface,                                               \
        direct                                           \
          analysis_driver = 'text_book'                  \

responses,                                               \
        num_objective_functions = 1                      \
        num_nonlinear_inequality_constraints = 2         \
        numerical_gradients                              \
          interval_type central                          \
          method_source dakota                           \
          fd_gradient_step_size = 1.0E-5                 \
        no_hessians
```

Figure 13.6: DAKOTA input file for the branch-and-bound strategy for solving MINLP optimization problems.

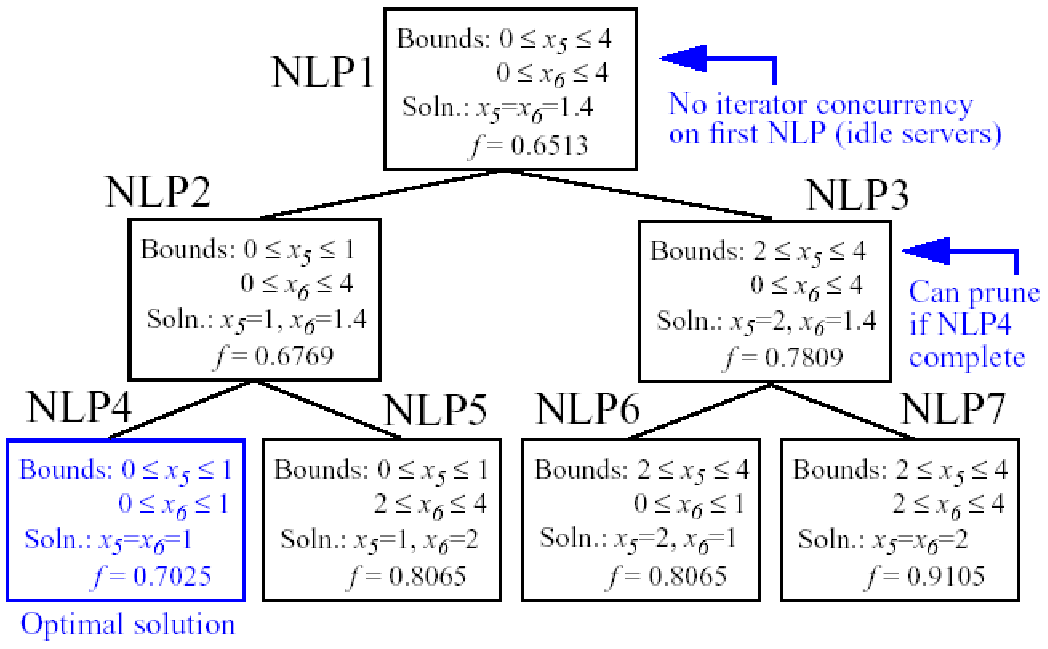

NLP1
Bounds: $0 \le x_5 \le 4$, $0 \le x_6 \le 4$
Soln.: $x_5 = x_6 = 1.4$
$f = 0.6513$
No iterator concurrency on first NLP (idle servers)
NLP2
Bounds: $0 \le x_5 \le 1$, $0 \le x_6 \le 4$
Soln.: $x_5 = 1$, $x_6 = 1.4$
$f = 0.6769$
NLP3
Bounds: $2 \le x_5 \le 4$, $0 \le x_6 \le 4$
Soln.: $x_5 = 2$, $x_6 = 1.4$
$f = 0.7809$
Can prune if NLP4 complete
NLP4
Bounds: $0 \le x_5 \le 1$, $0 \le x_6 \le 1$
Soln.: $x_5 = x_6 = 1$
$f = 0.7025$
Optimal solution
NLP5
Bounds: $0 \le x_5 \le 1$, $2 \le x_6 \le 4$
Soln.: $x_5 = 1$, $x_6 = 2$
$f = 0.8065$
NLP6
Bounds: $2 \le x_5 \le 4$, $0 \le x_6 \le 1$
Soln.: $x_5 = 2$, $x_6 = 1$
$f = 0.8065$
NLP7
Bounds: $2 \le x_5 \le 4$, $2 \le x_6 \le 4$
Soln.: $x_5 = x_6 = 2$
$f = 0.9105$

Figure 13.7: Branching history for example MINLP optimization problem.

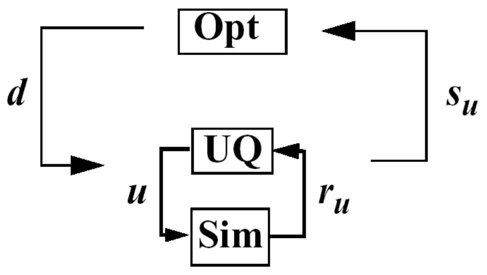


Figure 13.8: Formulation 1: Nested OUU.

are provided in [19].

13.6.1 Nested OUU

In the case of a nested approach, the optimization loop is the outer loop which seeks to optimize a nondeterministic quantity (e.g., minimize probability of failure). The uncertainty quantification (UQ) inner loop evaluates this nondeterministic quantity (e.g., computes the probability of failure) for each optimization function evaluation. Figure 13.8 depicts the nested OUU iteration where $\mathbf{d}$ are the design variables, $\mathbf{u}$ are the uncertain variables characterized by probability distributions, $\mathbf{r_u}(\mathbf{d}, \mathbf{u})$ are the response functions from the simulation, and $\mathbf{s_u}(\mathbf{d})$ are the statistics generated from the uncertainty quantification on these response functions.

Figure 13.9 shows a DAKOTA input file for a nested OUU example problem that is based on the textbook test problem. This input file is named `dakota_ouu1_tb.in` in the `/Dakota/test` directory. In this example, the objective function contains two probability of failure estimates, and an inequality constraint contains another probability of failure estimate. For this example, failure is defined to occur when one of the textbook response functions exceeds its threshold value. The strategy keyword block at the top of the input file identifies this as an OUU problem. The strategy keyword block is followed by the optimization specification, consisting of the optimization method, the continuous design variables, and the response quantities that will be used by the optimizer. The mapping matrices used for incorporating UQ statistics into the optimization response data are described in the DAKOTA Reference Manual [17]. The uncertainty quantification specification includes the UQ method, the uncertain variable probability distributions, the interface to the simulation code, and the UQ response attributes. As with other complex DAKOTA input files, the identification tags given in each keyword block can be used to follow the relationships among the different keyword blocks.

Latin hypercube sampling is used as the UQ method in this example problem. Thus, each evaluation of the response functions by the optimizer entails 50 Latin hypercube samples. In general, nested OUU studies can easily generate several thousand function evaluations and gradient-based optimizers may not perform well due to noisy or insensitive statistics resulting from under-resolved sampling. These observations motivate the use of surrogate-based approaches to OUU.

Other nested OUU examples in the `/Dakota/test` directory include `dakota_ouu1_tbch.in`, which adds an additional interface for including deterministic data in the textbook OUU problem, and `dakota_ouu1_cantilever.in`, which solves the cantilever OUU problem (see Section 20.5) with a nested approach. For each of these files, the "`1`" identifies formulation 1, which is short-hand for the nested approach.

13.6.2 Surrogate-Based OUU (SBOUU)

Surrogate-based optimization under uncertainty strategies can be effective in reducing the expense of OUU studies. Possible formulations include use of a surrogate model at the optimization level, at the uncertainty quantification level, or at both levels. These surrogate models encompass both data fit surrogates (at the optimization or UQ level) and model hierarchy surrogates (at the UQ level only). Figure 13.10 depicts the different surrogate-based formulations where $\hat{\mathbf{r}}_\mathbf{u}$ and $\hat{\mathbf{s}}_\mathbf{u}$ are approximate response functions and approximate response statistics, respectively, generated from the surrogate models.

SBOUU examples in the `/Dakota/test` directory include `dakota_sbouu2_tbch.in`, `dakota_sbouu3_tbch.in`, and `dakota_sbouu4_tbch.in`, which solve the textbook OUU problem, and `dakota_sbouu2_cantilever.in`, `dakota_sbouu3_cantilever.in`, and `dakota_sbouu4_cantilever.in`, which solve the cantilever OUU problem (see Section 20.5). For each of these files, the "2," "3," and "4" identify formulations 2, 3, and 4, which are short-hand for the "layered containing nested," "nested containing layered," and "layered containing nested containing layered" surrogate-based formulations, respectively. In general, the use of surrogates greatly reduces the computational expense of these OUU study. However, without restricting and verifying the steps in the approximate optimization cycles, weaknesses in the data fits can be exploited and poor solutions may be obtained. The need to maintain accuracy of results leads to the use of trust-region surrogate-based approaches.

13.6.3 Trust-Region Surrogate-Based OUU (TR-SBOUU)

The TR-SBOUU approach applies the trust region logic of deterministic SBO (see Section 13.6) to SBOUU. Trust-region verifications are applicable when surrogates are used at the optimization level, i.e., formulations 2 and 4. As a result of periodic verifications and surrogate rebuilds, these techniques are more expensive than SBOUU; however they are more reliable in that they maintain the accuracy of results. Relative to nested OUU (formulation 1), TR-SBOUU tends to be less expensive and less sensitive to initial seed and starting point.

SBOUU examples in the `/Dakota/test` directory include `dakota_trsbouu2_tbch.in` and `dakota_trsbouu4_tbch.in`, which solve the textbook OUU problem, and `dakota_trsbouu2_cantilever.in` and `dakota_trsbouu4_cantilever.in`, which solve the

```
strategy,                                                       \
        single_method                                   \
          method_pointer = ’OPTIM’

method,                                                                 \
        id_method = ’OPTIM’                                             \
        model_pointer = ’OPTIM_M’                                       \
        npsol_sqp                                                       \
          convergence_tolerance = 1.e-8

model,                                                                  \
        id_model = ’OPTIM_M’                                            \
        nested                                                          \
          variables_pointer  = ’OPTIM_V’                                \
          sub_method_pointer = ’UQ’                                     \
          responses_pointer  = ’OPTIM_R’                                \
          primary_response_mapping   = 0. 0. 1. 0. 0. 1. 0. 0. 0.       \
          secondary_response_mapping = 0. 0. 0. 0. 0. 0. 0. 0. 1.

variables,                                                      \
        id_variables = ’OPTIM_V’                                \
        continuous_design = 2                                   \
          cdv_initial_point    1.8    1.0                       \
          cdv_upper_bounds     2.164  4.0                       \
          cdv_lower_bounds     1.5    0.0                       \
          cdv_descriptor       ’d1’   ’d2’

responses,                                                      \
        id_responses = ’OPTIM_R’                                \
        num_objective_functions = 1                             \
        num_nonlinear_inequality_constraints = 1                \
        nonlinear_inequality_upper_bounds = .1                  \
        numerical_gradients                                     \
          method_source dakota                                  \
          interval_type central                                 \
          fd_gradient_step_size = 1.e-1                         \
        no_hessians

method,                                                         \
        id_method = ’UQ’                                        \
        model_pointer = ’UQ_M’                                  \
        nond_sampling,                                          \
          samples = 50 seed = 1 sample_type lhs                 \
          response_levels = 3.6e+11 1.2e+05 3.5e+05             \
          complementary distribution

model,                                                          \
        id_model = ’UQ_M’                                       \
        single                                                  \
          variables_pointer = ’UQ_V’                            \
          interface_pointer = ’UQ_I’                            \
          responses_pointer = ’UQ_R’

variables,                                                      \
        id_variables = ’UQ_V’                                   \
        continuous_design = 2                                   \
          cdv_descriptor       ’d1’   ’d2’                      \
        normal_uncertain = 2                                    \
          nuv_means             =  248.89, 593.33               \
          nuv_std_deviations    =   12.4,  29.7                 \
          nuv_descriptor        =  ’nuv1’  ’nuv2’               \
        uniform_uncertain = 2                                   \
          uuv_lower_bounds =  199.3,  474.63            \
          uuv_upper_bounds =  298.5,  712.                      \
          uuv_descriptor        =  ’uuv1’  ’uuv2’               \
        weibull_uncertain = 2                                   \
          wuv_alphas            =    12., 30.                   \
          wuv_betas             =   250., 590.                  \
          wuv_descriptor        =  ’wuv1’  ’wuv2’

interface,                                                      \
        id_interface = ’UQ_I’                                   \
        system asynch evaluation_concurrency = 5                \
          analysis_driver=        ’text_book_ouu’

responses,                                                      \
        id_responses = ’UQ_R’                                   \
        num_response_functions = 3                              \
        no_gradients                                            \
        no_hessians
```

Figure 13.9: DAKOTA input file for the nested OUU example.

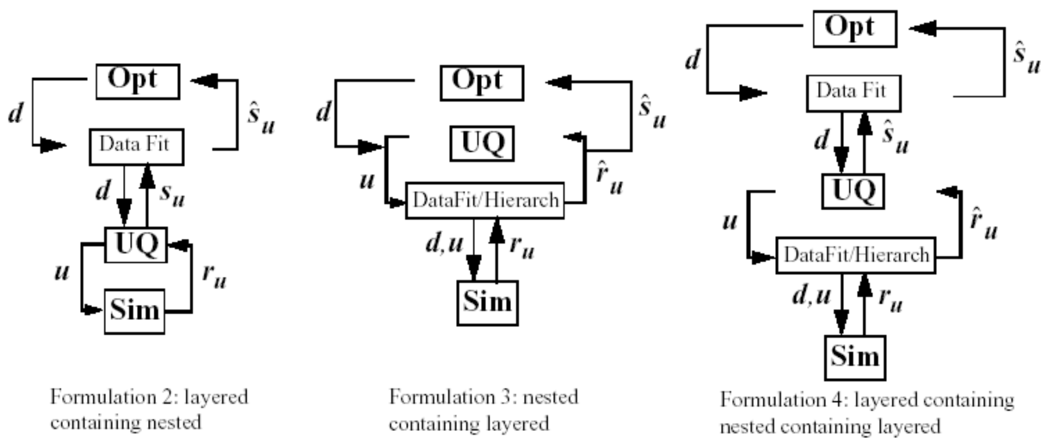


Figure 13.10: Formulations 2, 3, and 4 for Surrogate-based OUU.

cantilever OUU problem (see Section 20.5).

The TR-SBOUU algorithms are a subject of active research and development. Initial computational results for several example problems are available in [19].

13.7 Surrogate-Based Optimization (SBO)

In the surrogate-based optimization strategy (keyword: `surrogate_based_opt`) the optimization algorithm operates on a surrogate model instead of directly operating on the computationally expensive simulation model. The surrogate model can be formed from data samples and surface fit functions, or it can be a simplified version (e.g., coarsened finite element mesh) of the original simulation model. For either type of surrogate model, the SBO algorithm periodically checks the accuracy of the surrogate model against the original simulation model. The SBO strategy in DAKOTA can be implemented using heuristic rules (less expensive) or provably-convergent rules (more expensive). The heuristic SBO strategy is particularly effective on real-world engineering design problems that contain nonsmooth features (e.g., slope discontinuities, numerical noise) where gradient-based optimization methods often have trouble, and where the computational expense of the simulation precludes the use of nongradient-based methods.

13.7.1 SBO with Surface Fit Models

In SBO with surface fit functions, a sequence of optimization subproblems are evaluated, each of which is confined to a subset of the parameter space known as a “trust region.” Inside each trust region, DAKOTA’s data sampling methods are used to evaluate the response quantities at a small number (order 10^1 to 10^2) of design points. Next, multidimensional surface fitting is performed to create a surrogate function for each of the response quantities. Finally, optimization is performed using the surrogate functions in lieu of the actual response quantities, and the optimizer’s search is limited to the region inside the trust region bounds. A validation procedure is then applied to compare the predicted improvement in the response quantities to the actual improvement in the response quantities. Based on the results of this validation, the optimum design point is either accepted or rejected and the size of the trust region is either expanded, contracted, or left unchanged. The sequence of optimization subproblems continues until the SBO strategy convergence criteria are satisfied. More information on the data sampling methods is available in Chapter 9, and the surface fitting methods are described in Chapter 14.

Figure 13.11 shows a DAKOTA input file that implements surrogate-based optimization on Rosenbrock’s function. This input file is named `dakota_sbo_rosen.in` in the `/Dakota/test` directory. The strategy keyword block contains the SBO strategy keyword `surrogate_based_opt`, plus the commands for specifying the trust region size and scaling factors. The optimization portion of SBO is specified in the following keyword blocks for `method`, `variables`, `interface`, and `responses`. In SBO, the interface keyword block specifies the type of surface fit method on which the optimizer will operate. The data sampling portion of SBO is specified in an additional set of keyword blocks for `method`, `interface`, and `responses`. This example problem uses the Latin hypercube sampling method in the LHS software to select 10 design points in each trust region. (Note: to use Latin hypercube sampling from DDACE, swap the comment flags for the `nond_sampling` and `dace lhs` sections in the input file.) A single surrogate model is constructed for the objective function using a quadratic polynomial. The initial trust region is centered at the design point $(x_1, x_2) = (0.9, 0.9)$, and extends ± 0.4 from this point in the x_1 and x_2 coordinate directions.

If this input file is executed in DAKOTA, it will converge to the optimal design point at $(x_1, x_2) = (1, 1)$ in approximately 800 function evaluations. While this solution is correct, it is obtained at a much higher cost than a traditional gradient-based optimizer (e.g., see the results obtained from `dakota_rosenbrock.in`). The SBO strategy is not intended for use with smooth continuous optimization problems; gradient-based optimization is much more efficient for such applications. Rather, SBO is best-suited for the types of problems that occur in engineering design where the response quantities may be discontinuous, nonsmooth, or may have multiple local optima [34]. In these types of engineering design problems, traditional gradient-based optimizers often are ineffective. (For an example problem with multiple local optima, look in `/Dakota/test` for the file `dakota_sbo_sine_fcn.in` [35]).

A recently added capability for DAKOTA's SBO strategy is the incorporation of correction factors that improve the local accuracy of the surrogate models. The correction factors force the surrogate models to match the true function values, and possibly gradients and Hessians, at the center point of each trust region. Currently, DAKOTA supports either zeroth-, first-, or second-order accurate correction methods, each of which can be applied using either an additive, multiplicative, or combined additive/multiplicative function. The default behavior is that no correction factor is applied.

To visualize how these corrections are applied, consider two curves, $f_t(x)$ and $f_s(x)$, where $f_s(x)$ is the surrogate model for the true function $f_t(x)$. At the center point of each trust region, x_c, the correction factor approach creates a third function, $\hat{f}(x)$ that will be used by the optimizer. Note that in SBO without any correction factors, the optimizer operates directly on $f_s(x)$. For the `additive zeroth_order` method, the corrected function has the form $\hat{f}(x) = f_s(x) + [f_t(x_c) - f_s(x_c)]$. For the `multiplicative zeroth_order` method, the corrected function has the form $\hat{f}(x) = \alpha(x_c) f_s(x)$, where $\alpha(x_c) = f_t(x_c)/f_s(x_c)$. The `additive first_order` correction method, which is based on the work of Lewis and Nash [48], has the form $\hat{f}(x) = f_s(x) + [f_t(x_c) - f_s(x_c)] + [\nabla f_t(x_c) - \nabla f_s(x_c)]^T (x - x_c)$. The `multiplicative first_order` correction method, which is based on the work of Chang, et al., [9] and Alexandrov, et al, [1], has the form $\hat{f}(x) = \beta(x) f_s(x)$ and uses a scaling function, $\beta(x)$, that is computed using a first-order Taylor Series expansion $\beta(x) = \alpha(x_c) + \nabla\alpha(x_c)^T (x - x_c)$.

It should be noted that in both first order correction methods, the function $\hat{f}(x)$ matches the function value and gradients of $f_t(x)$ at $x = x_c$. This property is necessary in proving that the first order-corrected SBO algorithms are provably convergent to a local minimum of $f_t(x)$. However, the first order correction methods are significantly more expensive than the zeroth order correction methods, since the first order methods require computing both $\nabla f_t(x_c)$ and $\nabla f_s(x_c)$. When the SBO strategy is used with either of the zeroth order correction methods, or with no correction method, convergence is not guaranteed to a local minimum of $f_t(x)$. That is, the SBO strategy becomes a heuristic optimization algorithm. From a mathematical point of view this is undesirable, but as a practical matter, the heuristic variants of SBO are often effective in finding local minima.

`Usage guidelines:` As of April 2003, the DAKOTA team is continuing to test the surface fit SBO strategy using the various correction factor methods. Thus, no clear-cut guidelines are available. However, the user should consider the following observations:

```
# test file with a specific test.  The  is used to designate lines


strategy,                                               \
        surrogate_based_opt                             \
        tabular_graphics_data                           \
        max_iterations = 10000                          \
        opt_method_pointer = ’NLP’                      \
        trust_region                                    \
          initial_size = 0.10                           \
          minimum_size = 1.0e-6                         \
          contract_threshold = 0.25             \
          expand_threshold   = 0.75             \
          contraction_factor = 0.50                     \
          expansion_factor   = 1.50

method,                                                 \
        id_method = ’NLP’                               \
        model_pointer = ’SURROGATE’                     \
        conmin_frcg,                                    \
          max_iterations = 50,                          \
          convergence_tolerance = 1e-8

model,                                                          \
        id_model = ’SURROGATE’                                  \
        surrogate global                                \
          responses_pointer = ’SURROGATE_RESP’                  \
          dace_method_pointer = ’SAMPLING’              \
          correction additive zeroth_order              \
          polynomial quadratic                                  \

variables,                                              \
        continuous_design = 2                           \
          cdv_initial_point   -1.2      1.0             \
          cdv_lower_bounds    -2.0     -2.0             \
          cdv_upper_bounds     2.0      2.0             \
          cdv_descriptor       ’x1’     ’x2’

responses,                                              \
        id_responses = ’SURROGATE_RESP’                 \
        num_objective_functions = 1                     \
        numerical_gradients                             \
          method_source dakota                          \
          interval_type central                         \
          fd_gradient_step_size = 1.e-6                 \
        no_hessians

method,                                                 \
        id_method = ’SAMPLING’                          \
        model_pointer = ’TRUTH’                         \
        nond_sampling                                   \
          samples = 10                          \
          seed = 531                                    \
          sample_type lhs                               \
          all_variables

model,                                                  \
        id_model = ’TRUTH’                              \
        single                                          \
          interface_pointer = ’TRUE_FN’                 \
          responses_pointer = ’TRUE_RESP’

interface,                                              \
        direct                                          \
        id_interface = ’TRUE_FN’                        \
          analysis_driver = ’rosenbrock’

responses,                                              \
        id_responses = ’TRUE_RESP’                      \
        num_objective_functions = 1                     \
        no_gradients                            \
        no_hessians                     \
```

Figure 13.11: DAKOTA input file for the surrogate-based optimization example.

- Both the `additive zeroth_order` and `multiplicative zeroth_order` correction methods are "free" since they use values of $f_t(x_c)$ that are normally computed by the SBO strategy.
- The use of either the `additive first_order` method or the `multiplicative first_order` method does not necessarily improve the rate of convergence of the SBO algorithm.
- When using the first order correction methods, the `TRUE_FCN_GRAD` response keywords must be modified (see bottom of Figure 13.11) to allow either analytic or numerical gradients to be computed. This provides the gradient data needed to compute the correction function.
- For many computationally expensive engineering optimization problems, gradients often are too expensive to obtain or are discontinuous (or may not exist at all). In such cases the heuristic SBO algorithm has been an effective approach at identifying optimal designs [34].

13.7.2 SBO with Multifidelity Models

SBO can also be applied with multifidelity, or hierarchical, models, i.e., where one has available both a high-fidelity computational model and a low-fidelity computational model. This situation can occur when the low-fidelity model neglects some physical phenomena (e.g., viscosity, heat transfer, etc.) that are included in the high-fidelity model, or when the low-fidelity model has a lower resolution computational mesh than the high-fidelity model. In many cases, the low-fidelity model can serve as a surrogate for the high-fidelity model during the optimization process. Thus, the low-fidelity model can be used in SBO in a manner similar to the use of surface fit models described in Section 13.7.1. A key difference in SBO with hierarchical surrogates is that a design of experiments using the high-fidelity model is not required; rather high-fidelity evaluations are only needed at the center of the current trust-region and the predicted optimum point in order to correct the low-fidelity model and verify improvement, respectively. Another difference is that one of the four types of correction described in Section 13.7.1 is required for SBO with multifidelity models.

A multifidelity test problem named `dakota_sbo_hierarchical.in` is available in `/Dakota/test` to demonstrate this SBO approach. This test problem uses the Rosenbrock function as the high fidelity model and a function named "lf_rosenbrock" as the low fidelity model. Here, lf_rosenbrock is a variant of the Rosenbrock function (see `/Dakota/test/lf_rosenbrock.C` for formulation) with the minimum point at $(x_1, x_2) = (0.80, 0.44)$, whereas the minimum of the original Rosenbrock function is $(x_1, x_2) = (1, 1)$. Of the four correction approaches, only `additive first_order` is successful at reliably locating the high-fidelity minimum at $(x_1, x_2) = (1, 1)$ from arbitrary starting points. This likely results from the fact that the low- and high-fidelity Rosenbrock functions have similar contours and the `additive first_order` correction induces less skewing in the contours of the low fidelity model.

Chapter 14

Surface Fitting Methods

14.1 Overview

DAKOTA contains several types of surface fitting methods that can be used with optimization and uncertainty quantification methods and strategies such as surrogate-based optimization and optimization under uncertainty. These are: polynomial models (linear, quadratic, and cubic), first-order Taylor series expansion, kriging spatial interpolation, artificial neural networks, and multivariate adaptive regression splines. All of these surface fitting methods can be applied to problems having an arbitrary number of design parameters. However, surface fitting methods usually are practical only for problems where there are a small number of parameters (e.g., a maximum of somewhere in the range of 30-50 design parameters). The mathematical models created by surface fitting methods have a variety of names in the engineering community. These include surrogate models, meta-models, approximation models, and response surfaces. For this manual, the terms surface fit model and surrogate model are used.

14.2 Procedures for Surface Fitting

The surface fitting process consists of three steps: (1) selection of a set of design points, (2) evaluation of the true response quantities (e.g., from a user-supplied simulation code) at these design points, and (3) using the response data to solve for the unknown coefficients (e.g., polynomial coefficients, neural network weights, kriging correlation factors) in the surface fit model. In cases where there is more than one response quantity (e.g., an objective function plus one or more constraints), then a separate surface is built for each response quantity. Currently, the surface fit models are built using only 0^{th}-order information (function values only), although extensions to using higher-order information (gradients and Hessians) are possible. Each surface fitting method employs a different numerical method for computing its internal coefficients. For example, the polynomial surface uses a least-squares approach that employs a singular value decomposition to compute the polynomial coefficients, whereas the kriging surface uses Maximum Likelihood Estimation to compute its correlation coefficients. More information on the numerical methods used in the surface fitting codes is provided in the DAKOTA Developers Manual [18].

The set of design points that is used to construct a surface fit model is generated using either the DDACE software package [64] or the LHS software package [45]. These packages provide a variety of sampling methods including Monte Carlo (random) sampling, Latin hypercube sampling, orthogonal array sampling, central composite design sampling, and Box-Behnken sampling. More information on these software packages is provided in Chapter 9.

14.3 Linear, Quadratic, and Cubic Polynomial Models

Linear, quadratic, and cubic polynomial models are available in DAKOTA. The form of the linear polynomial model is

$$\hat{f}(\mathbf{x}) \approx c_0 + \sum_{i=1}^{n} c_i x_i \tag{14.1}$$

the form of the quadratic polynomial model is:

$$\hat{f}(\mathbf{x}) \approx c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} c_{ij} x_i x_j \tag{14.2}$$

and the form of the cubic polynomial model is:

$$\hat{f}(\mathbf{x}) \approx c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} c_{ij} x_i x_j + \sum_{i=1}^{n} \sum_{j \geq i}^{n} \sum_{k \geq j}^{n} c_{ijk} x_i x_j x_k \tag{14.3}$$

In all of the polynomial models, $\hat{f}(\mathbf{x})$ is the response of the polynomial model; the x_i, x_j, x_k terms are the components of the n-dimensional design parameter values; the c_0 , c_i , c_{ij} , c_{ijk} terms are the polynomial coefficients, and n is the number of design parameters. The number of coefficients, n_c, depends on the order of polynomial model and the number of design parameters. For the linear polynomial:

$$n_{c_{linear}} = n + 1 \tag{14.4}$$

for the quadratic polynomial:

$$n_{c_{quad}} = \frac{(n+1)(n+2)}{2} \tag{14.5}$$

and for the cubic polynomial:

$$n_{c_{cubic}} = \frac{(n^3 + 6n^2 + 11n + 6)}{6} \tag{14.6}$$

There must be at least n_c data samples in order to form a fully determined linear system and solve for the polynomial coefficients. In DAKOTA, a least-squares approach involving a singular value decomposition numerical method is applied to solve the linear system.

The utility of the polynomial models stems from two sources: (1) over a small portion of the parameter space, a low-order polynomial model is often an accurate approximation to the true data trends, and (2) the least-squares procedure provides a surface fit that smooths out noise in the data. For this reason, the surrogate-based optimization strategy often is successful when using polynomial models, particularly quadratic models. However, a polynomial surface fit may not be the best choice for modeling data trends over the entire parameter space, unless it is known a priori that the true data trends are close to linear, quadratic, or cubic. See [54] for more information on polynomial models.

14.4 First-order Taylor Series Models

The first-order Taylor Series model is purely a local approximation method. That is, it provides local trends in the vicinity of a single point in parameter space. The form of the Taylor Series model is

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x}_0) + (\nabla_x f|_{\mathbf{x}=\mathbf{x}_0})^T (\mathbf{x} = \mathbf{x}_0) \tag{14.7}$$

where $\mathbf{x}_0$ is the current point in n-dimensional parameter space, $f(\mathbf{x}_0)$ is the computed response value at the current point, and $\nabla_x f|_{\mathbf{x}=\mathbf{x}_0}$ is the computed response gradient at the current point.

In general, the Taylor Series model is accurate only in the region of parameter space that is close to $\mathbf{x}_0$. While the accuracy is limited, the Taylor Series model has the correct value and gradient at the point $\mathbf{x}_0$. This first-order consistency is useful in provably-convergent surrogate-based optimization. The other surface fitting methods do not use gradient information directly in their models, and these methods rely on an external correction procedure in order to satisfy the consistency requirements of provably-convergent SBO.

14.5 Kriging Spatial Interpolation Models

The kriging method uses techniques developed in the geostatistics and spatial statistics communities ([11], [47]) to produce smooth, C^2-continuous surface fit models of the response values from a set of data points. The form of the kriging model is

$$\hat{f}(\mathbf{x}) \approx \beta + \mathbf{r}^T \mathbf{R}^{-1} (\mathbf{f} - \beta \mathbf{e}) \tag{14.8}$$

where $\mathbf{x}$ is the current point in n-dimensional parameter space; is the estimate of the mean response value, r is the correlation vector of terms between $\mathbf{x}$ and the data points, $\mathbf{R}$ is the correlation matrix for all of the data points, $\mathbf{f}$ is the vector of response values, and $\mathbf{e}$ is a vector with all values set to one. The terms in the correlation vector and matrix are computed using a Gaussian correlation function and are dependent on an n-dimensional vector of correlation parameters, $\Theta = \{\theta_1, \ldots, \theta_n\}$. In DAKOTA, a Maximum Likelihood Estimation procedure is performed to compute the correlation parameters for the kriging model. More detail on this kriging approach may be found in [36].

The kriging interpolation model is a nonparametric surface fitting approach. That is, the kriging surface does not assume that there is an underlying trend in the response data. This is in contrast to the quadratic polynomial model and the linear Taylor Series model. Since the kriging model is nonparametric, it can be used to model surfaces with slope discontinuities along with multiple local minima and maxima. Kriging interpolation is useful for both SBO and OUU, as well as for studying the global response value trends in the parameter space. This surface fitting method can be constructed using a minimum of $n_{c_{linear}}$ design points, but it is recommended to use at least $n_{c_{quad}}$ design points when possible (refer to Section 14.3 for n_c definitions).

The kriging model is guaranteed to pass through all of the response data values that are used to construct the model. Generally, this is a desirable feature. However, if there is considerable numerical noise in the response data, then a surface fitting method that provides some data smoothing (e.g., quadratic polynomial, MARS) may be a better choice for SBO and OUU applications. Another feature of the kriging model is that the predicted response values, $\hat{f}(\mathbf{x})$, decay to the mean value, β, when $\mathbf{x}$ is far from any of the data points from which the kriging model was constructed (i.e., when the model is used for extrapolation). This is neither a positive nor a negative aspect of kriging, but rather a different behavior than is exhibited by the other surface fitting methods. One drawback to the kriging model is that data points in close proximity lead to ill-conditioning in the numerical procedure and the kriging software will terminate if such a situation occurs. For this reason, the user is advised to avoid sample reuse (`reuse_samples = region` and `reuse_samples = all` specifications) when performing surrogate-based optimization.

14.6 Artificial Neural Network (ANN) Models

The ANN surface fitting method in DAKOTA employs a stochastic layered perceptron (SLP) artificial neural network based on the direct training approach of Zimmerman [72]. The SLP ANN method is designed to have a lower training cost than traditional ANNs. This is a useful feature for SBO and OUU where new ANNs are constructed many times during the optimization process (i.e., one ANN for each response function, and new ANNs for each optimization iteration). The form of the SLP ANN model is

$$\hat{f}(\mathbf{x}) \approx \tanh(\tanh((\mathbf{x}\mathbf{A}_0 + \theta_0)\mathbf{A}_1 + \theta_1)) \tag{14.9}$$

where $\mathbf{x}$ is the current point in n-dimensional parameter space, and the terms $\mathbf{A}_0, \theta_0, \mathbf{A}_1, \theta_1$ are the matrices and vectors that correspond to the neuron weights and offset values in the ANN model. These terms are computed during the ANN training process, and are analogous to the polynomial coefficients in a quadratic surface fit. A singular value decomposition method is used in the numerical methods that are employed to solve for the weights and offsets.

The SLP ANN is a non parametric surface fitting method. Thus, along with kriging and MARS, it can be used to model data trends that have slope discontinuities as well as multiple maxima and minima. However, unlike kriging, the ANN surface is not guaranteed to exactly match the response values of the data points from which it was constructed. This ANN can be used with SBO and OUU strategies. As with kriging, this ANN can be constructed from fewer than $n_{c_{quad}}$ data points, however, it is a good rule of thumb to use at least $n_{c_{quad}}$ data points when possible.

14.7 Multivariate Adaptive Regression Spline (MARS) Models

This surface fitting method uses multivariate adaptive regression splines from the MARS3.5 package [27] developed at Stanford University. Currently, access to the MARS software is provided through the DDACE package [64].

The form of the MARS model is based on the following expression:

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} a_m B_m(\mathbf{x}) \tag{14.10}$$

where the a_m are the coefficients of the truncated power basis functions B_m, and M is the number of basis functions. The MARS software partitions the parameter space into subregions, and then applies forward and backward regression methods to create a local surface model in each subregion. The result is that each subregion contains its own basis functions and coefficients, and the subregions are joined together to produce a smooth, C^2-continuous surface model.

MARS is a nonparametric surface fitting method and can represent complex multimodal data trends. The regression component of MARS generates a surface model that is not guaranteed to pass through all of the response data values. Thus, like the quadratic polynomial model, it provides some smoothing of the data. The MARS reference material does not indicate the minimum number of data points that are needed to create a MARS surface model. However, in practice it has been found that at least $n_{c_{quad}}$, and sometimes as many as 2 to 4 times $n_{c_{quad}}$, data points are needed to keep the MARS software from terminating. Provided that sufficient data samples can be obtained, MARS surface models can be useful in SBO and OUU applications, as well as in the prediction of global trends throughout the parameter space.

Chapter 15

Parallel Computing

15.1 Overview

Parallel computers within the Department of Energy national laboratories have exceeded ten trillion floating point operations per second (10 TeraFLOPS) and are expected to achieve 100 TeraFLOPS in the near future. This performance is achieved through the use of massively parallel (MP) processing ($O[10^3 - 10^4]$) processors). In order to harness the power of these machines for performing design, parallel optimization approaches are needed which are scalable on thousands of processors. To understand the possibilities, it is instructive to first categorize the opportunities for exploiting parallelism into four main areas [21], consisting of coarse-grained and fine-grained parallelism opportunities within algorithms and their function evaluations:

1. *Algorithmic coarse-grained parallelism*: This parallelism involves the concurrent execution of independent function evaluations, where a "function evaluation" is defined as a data request from an algorithm (which may involve value, gradient, and Hessian data from multiple objective and constraint functions). This concept can also be extended to the concurrent execution of multiple "iterators" within a "strategy." Examples of algorithms containing coarse-grained parallelism include:

    - *Gradient-based algorithms*: finite difference gradient evaluations, speculative optimization, parallel line search.
    - *Nongradient-based algorithms*: genetic algorithms (GAs), pattern search (PS), Monte Carlo sampling.
    - *Approximate methods*: design of computer experiments for building response surface approximations.
    - *Concurrent-iterator strategies*: optimization under uncertainty, branch and bound, multi-start local search, Pareto set optimization, island-model GAs.

2. *Algorithmic fine-grained parallelism*: This involves computing the basic computational steps of an optimization algorithm (i.e., the internal linear algebra) in parallel. This is primarily of interest in large-scale optimization problems and simultaneous analysis and design (SAND).

3. *Function evaluation coarse-grained parallelism*: This involves concurrent computation of separable parts of a single function evaluation. This parallelism can be exploited when the evaluation of the response data set requires multiple independent simulations (e.g. multiple loading cases or operational environments) or multiple dependent analyses where the coupling is applied at the optimizer level (e.g., the individual discipline feasible formulation [12]).

4. *Function evaluation fine-grained parallelism*: This involves parallelization of the solution steps within a single analysis code. The DOE laboratories have developed parallel analysis codes in the areas of nonlinear mechanics, structural dynamics, heat transfer, computational fluid dynamics, shock physics, and many others.

By definition, coarse-grained parallelism requires very little inter-processor communication and is therefore "embarrassingly parallel," meaning that there is little loss in parallel efficiency due to communication as the number of processors increases. However, it is often the case that there are not enough separable computations on each algorithm cycle to utilize the thousands of processors available on MP machines. For example, a thermal safety application [24] demonstrated this limitation with a pattern search optimization in which the maximum speedup exploiting *only* coarse-grained algorithmic parallelism was shown to be severely limited by the size of the design problem (coordinate pattern search has at most $2n$ independent evaluations per cycle for n design variables).

Fine-grained parallelism, on the other hand, involves much more communication among processors and care must be taken to avoid the case of inefficient machine utilization in which the communication demands among processors outstrip the amount of actual computational work to be performed. For example, a chemically-reacting flow application [21] illustrated this limitation for a simulation of fixed size in which it was shown that, while simulation run time did monotonically decrease with increasing number of processors, the relative parallel efficiency $\hat{E}$ of the computation for fixed model size decreased rapidly (from $\hat{E} \approx 0.8$ at 64 processors to $\hat{E} \approx 0.4$ at 512 processors). This was due to the fact that the total amount of computation was approximately fixed, whereas the communication demands were increasing rapidly with increasing numbers of processors. Therefore, there is a practical limit on the number of processors that can be employed for fine-grained parallel simulation of a particular model size, and only for extreme model sizes ("heroic-scale") can thousands of processors be efficiently utilized in studies exploiting fine-grained parallelism alone.

These limitations point us to the exploitation of multiple levels of parallelism, in particular the combination of coarse-grained and fine-grained approaches. DAKOTA supports a total of three tiers of scheduling and four levels of parallelism which, in combination, can minimize efficiency losses and achieve near linear scaling on MP computers. The four levels are:

- concurrent iterators within a strategy (scheduling performed by DAKOTA)
- concurrent function evaluations within each iterator (scheduling performed by DAKOTA)
- concurrent analyses within each function evaluation (scheduling performed by DAKOTA)
- multiprocessor analyses (work distributed by the parallel analysis code)

for which the first two are classified as algorithmic coarse-grained parallelism, the third is function evaluation coarse-grained parallelism, and the fourth is function evaluation fine-grained parallelism. Algorithmic fine-grained parallelism is not currently supported, although the development of large-scale parallel SAND techniques is a current research direction [5].

A particular application may support one or more of these parallelism types, and DAKOTA provides for convenient selection and combination of each of the supported levels. If multiple types of parallelism can be exploited, then the question may arise as to how the amount of parallelism at each level should be selected so as to maximize the overall parallel efficiency of the study. For performance analysis of multilevel parallelism formulations and detailed discussion of these issues, refer to [22]. *In general, it is recommended that the user employ DAKOTA's automatic parallelism configuration facilities, as these utilize the recommendations from the aforementioned paper*.

While development of techniques for high end MP computers is a primary research driver, it is important to note that DAKOTA's parallel facilities support a broad range of hardware and are equally applicable to parallel processing on networks of workstations (NOWs) or desktop multiprocessors. Given the reduced scale in these cases, it is more common to exploit only one of the levels of parallelism; however, this can still be quite effective in reducing the time to obtain a solution.

In the following sections, the parallel algorithms available in this DAKOTA release are listed followed by descriptions of the software components which enable parallelism, approaches for utilizing these components, and input specification and execution details for running parallel DAKOTA studies.

15.2 Parallel Algorithms

In DAKOTA Version 3.1, the following iterators and strategies support algorithmic coarse-grained parallelism.

15.2.1 Parallel iterators

- Gradient-based optimizers: CONMIN, NPSOL, DOT, and OPT++ can all exploit parallelism through the use of DAKOTA's native finite differencing routine (selected with `method_source dakota` in the responses specification), which will perform concurrent evaluations for each of the parameter offsets. For `n` variables, forward differences result in an $n + 1$ concurrency and central differences result in a $2n+1$ concurrency. In addition, CONMIN, DOT, and OPT++ can use speculative gradient techniques [8] to obtain better parallel load balancing. By speculating that the gradient information associated with a given line search point will be used later and computing the gradient information in parallel at the same time as the function values, the concurrency during the gradient evaluation and line search phases can be balanced. NPSOL does not use speculative gradients since this approach is superseded by NPSOL's gradient-based line search in user-supplied derivative mode.
- Nongradient-based optimizers: most COLINY methods support parallelism. Serial COLINY methods include Solis-Wets (`coliny_solis_wets`) and certain `exploratory_moves` options (`adaptive_pattern` and `multi_step`) in pattern search (`coliny_pattern_search`). PDS within OPT++ (`optpp_pds`) is also currently serial due to limitations in the OPT++ interface.
- Least squares methods: in an identical manner to the gradient-based optimizers, NLSSOL and Gauss-Newton can exploit parallelism through the use of DAKOTA's native finite differencing routine. In addition, Gauss-Newton can use speculative gradient techniques to obtain better parallel load balancing. NLSSOL does not use speculative gradients since this approach is superseded by NLSSOL's gradient-based line search in user-supplied derivative mode.
- Parameter studies: all parameter study methods (`vector`, `list`, `centered`, and `multidim`) support parallelism. These methods avoid internal synchronization points, so all evaluations are available for concurrent execution.
- Design of experiments: all `dace` methods (`grid`, `random`, `oas`, `lhs`, `oa_lhs`, `box_behnken`, and `central_composite`) support parallelism.
- Uncertainty quantification: all nondeterministic methods (`nond_sampling`, `nond_analytic_reliability`, and `nond_polynomial_chaos`) support parallelism. In the case of `nond_analytic_reliability`, gradient-based optimization is involved and parallelism can be exploited through the use of DAKOTA's native finite differencing routine.

15.2.2 Parallel strategies

Certain strategies support concurrency in multiple iterator executions. Currently, the strategies which can exploit this level of parallelism are:

- Branch and bound
- Pareto-set optimization
- Multi-start iteration

In the branch and bound case, the available iterator concurrency grows as the tree develops more branches, so some of the iterator servers may be idle in the initial phases. Pareto-set and multi-start, however, have a fixed set of jobs to perform and should exhibit good load balancing. In a future release, optimization under uncertainty will be added to the strategies which support concurrent iterator parallelism.

15.3 Local Simulation Invocation Components

This section describes software components which manage simulation invocations local to a processor. These invocations may be either synchronous (i.e., blocking) or asynchronous (i.e., nonblocking). Synchronous evaluations proceed one at a time with the evaluation running to completion before control is returned to DAKOTA. Asynchronous evaluations are initiated such that control is returned to DAKOTA immediately, prior to evaluation completion, thereby allowing the initiation of additional evaluations which will execute concurrently.

The synchronous local invocation capabilities are used to provide serial execution on a single processor and also to provide function evaluations local to a processor within DAKOTA's message-passing schedulers. The asynchronous local invocation capabilities can be used by themselves to provide a simple parallelism which relies on external means to assign jobs to processors, or they can be combined with DAKOTA's message-passing schedulers to provide a hybrid parallelism. Refer to Section 15.5 for additional details.

In most cases, blocking schedulers are used for the management of sets of asynchronous local evaluations, in which all jobs in the queue are completed before exiting the scheduler and returning the set of results to the algorithm. Nonblocking asynchronous local schedulers are also available for the case of fully asynchronous algorithms which do not contain synchronization points (e.g., the APPS algorithm). In this case, jobs may come and go from the queue without the enforcement of a hard synchronization point.

DAKOTA supports three approaches to local simulation invocation based on the direct function, system call, and fork application interfaces. For each of these cases, an input filter, one or more analysis drivers, and an output filter make up the interface, as described in Section 5.6.

15.3.1 Direct function synchronization

The direct function capability may be used synchronously. Synchronous operation of the direct function application interface involves a standard procedure call to the input filter, if present, followed by calls to one or more simulations, followed by a call to the output filter, if present. Each of these components must be linked as functions within DAKOTA. Control does not return to the calling code until the evaluation is completed and the response object has been populated.

Asynchronous operation will be supported in the future and will involve the use of multithreading (e.g., POSIX threads) to accomplish multiple simultaneous simulations. When spawning a thread (e.g., using `pthread_create`), control returns to the calling code after the simulation is initiated. In this way, multiple threads can be created simultaneously. An array of responses corresponding to the multiple threads of execution would then be recovered in a synchronize operation (e.g., using `pthread_join`).

15.3.2 System call synchronization

The system call capability may be used synchronously or asynchronously. In both cases, the `system` utility from the standard C library is used. Synchronous operation of the system call application interface involves spawning the system call (containing the filters and analysis drivers bound together with parentheses and semi-colons) in the foreground. Control does not return to the calling code until the simulation is completed and the response file has been written. In this case, the possibility of a race condition (see below) does not exist and any errors during response recovery will cause an immediate abort of the DAKOTA process (note: detection of the string "fail" is not a response recovery error; see Chapter 19).

Asynchronous operation involves spawning the system call in the background, continuing with other tasks (e.g., spawning other system calls), periodically checking for process completion, and finally retrieving the results. An array of responses corresponding to the multiple system calls is recovered in a synchronize operation.

In this synchronize operation, completion of a function evaluation is detected by testing for the existence of the evaluation's results file using the `stat` utility [46]. Care must be taken when using asynchronous system calls since they are prone to the race condition in which the results file passes the existence test but the recording of the function evaluation results in the file is incomplete. In this case, the read operation performed by DAKOTA will result in an error due to an incomplete data set. In order to address this problem, DAKOTA contains exception handling which allows for a fixed number of response read failures per asynchronous system call evaluation. The number of allowed failures must have a limit, so that an actual response format error (unrelated to the race condition) will eventually abort the system. Therefore, to reduce the possibility of exceeding the limit on allowable read failures, *the user's interface should minimize the amount of time an incomplete results file exists in the directory where its status is being tested*. This can be accomplished through two approaches: (1) delay the creation of the results file until the simulation computations are complete and all of the response data is ready to be written to the results file, or (2) perform the simulation computations in a subdirectory, and as a last step, move the completed results file into the main working directory where its existence is being queried.

If concurrent simulations are executing in a shared disk space, then care must be taken to maintain independence of the simulations. In particular, the parameters and results files used to communicate with DAKOTA, as well as any other files used by this simulation, must be protected from other files of the same name used by the other concurrent simulations. With respect to the parameters and results files, these files may be made unique through the use of the `file_tag` option (e.g., `params.in.1`, `results.out.1`, etc.) or the default UNIX temporary file option (e.g., `/var/tmp/aaa0b2Mfv`, etc.). However, if additional simulation files must be protected (e.g., `model.i`, `model.o`, `model.g`, `model.e`, etc.), then an effective approach is to create a tagged working subdirectory for each simulation instance. Section 16.1 provides an example system call interface that demonstrates both the use of tagged working directories and the relocation of completed results files to avoid the race condition.

15.3.3 Fork synchronization

The fork capability is quite similar to the system call; however, it has the advantage that asynchronous fork invocations can avoid the results file race condition that may occur with asynchronous system calls. The fork interface invokes the filters and analysis drivers using the `fork` and `exec` family of functions, and completion of these processes is detected using the `wait` family of functions. Since `wait` is based on a process id handle rather than a file existence test, an incomplete results file is not an issue.

Depending on the platform, the fork application interface executes either a `vfork` or a `fork` call. These calls generate a new child process with its own UNIX process identification number, which functions as a copy of the parent process (dakota). The `execvp` function is then called by the child process, causing it to be replaced by the analysis driver or filter. For synchronous operation, the parent dakota process then awaits completion of the forked child process through a blocking call to `waitpid`. On most platforms, the `fork/exec` procedure is efficient since it operates in a copy-on-write mode, and no copy of the parent is actually created. Instead, the parents address space is borrowed until the `exec` function is called.

The `fork/exec` behavior for asynchronous operation is similar to that for synchronous operation, the only difference being that dakota invokes multiple simulations through the `fork/exec` procedure prior to recovering response results for these jobs using the `wait` function. The combined use of `fork/exec` and `wait` functions in asynchronous mode allows the scheduling of a specified number of concurrent function evaluations and/or concurrent analyses.

15.4 Message Passing Components

DAKOTA uses a “single program-multiple data” (SPMD) parallel programming model. It uses message-passing routines from the Message Passing Interface (MPI) standard [39], [62] to communicate data between processors. The SPMD designation simply denotes that the same DAKOTA executable is loaded on all processors during the parallel invocation. This differs from the MPMD model (“multiple program-multiple data”) which would have the DAKOTA executable on one or more processors communicating directly with simulator executables on other processors. The MPMD model has some advantages, but heterogeneous executable loads are not supported by all parallel environments. Moreover, the MPMD model requires simulation code intrusion on the same order as conversion to a subroutine, so the subroutine conversion in a direct-linked SPMD model is preferred.

15.4.1 Partitioning of levels

DAKOTA uses MPI communicators to identify groups of processors. The global `MPI_COMM_WORLD` communicator provides the total set of processors allocated to the DAKOTA run. `MPI_COMM_WORLD` can be partitioned into new intra-communicators which each define a set of processors to be used for a multiprocessor server. Each of these servers may be further partitioned to nest one level of parallelism within the next. At the lowest parallelism level, these intra-communicators can be passed into a simulation for use as the simulation’s computational context, provided that the simulation has been designed, or can be modified, to be modular on a communicator (i.e., it does not assume ownership of `MPI_COMM_WORLD`). New intra-communicators are created with the `MPI_Comm_split` routine, and in order to send messages between these intra-communicators, new inter-communicators are created with calls to `MPI_Intercomm_create`. To minimize overhead, DAKOTA creates new intra- and inter-communicators only when the parent communicator provides insufficient context for the scheduling at a particular level. In addition, communicator partitions can be reallocated multiple times. This enables dynamic repartitioning for a strategy that manages multiple iterators and models (e.g., four 256 processor servers could be used for iteration on a lower fidelity model, followed by two 512 processor servers for subsequent iteration on a higher fidelity model). In DAKOTA, communicator partitioning schemes are allocated and deallocated for each iterator/model pair within those strategies for which multi-fidelity models may be present (e.g., the multilevel optimization strategy described in Section 13.2).

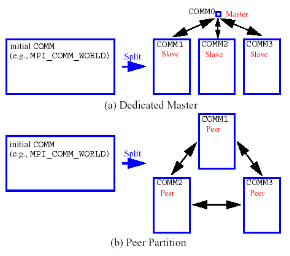


Figure 15.1: Communicator partitioning models.

Each tier within DAKOTA’s nested parallelism hierarchy can use either of two processor partitioning models: a “dedicated master” partitioning in which a single processor is dedicated to scheduling operations and the remaining processors are split into server partitions, or a “peer partition” approach in which the loss of a processor to scheduling is avoided. These models are depicted in Figure 15.1. The peer partition is desirable since it utilizes all processors for computation; however, it requires either the use of sophisticated mechanisms for distributed scheduling or a problem for which static scheduling of concurrent work performs well (see *Scheduling within levels* below). To recursively partition the subcommunicators of Figure 15.1, `COMM1/2/3` in the dedicated master or peer partition case would be further subdivided using the appropriate partitioning model for the next lower level of parallelism.

15.4.2 Scheduling within levels

The following scheduling approaches are available within each level:

- *Self-scheduling*: in the dedicated master model, the master processor manages a single processing queue and maintains a prescribed number of jobs (usually one) active on each slave. Once a slave server has completed a job and returned its results, the master assigns the next job to this slave. Thus, the slaves themselves determine the schedule through their job completion speed. This provides a simple dynamic scheduler in that heterogeneous processor speeds and/or job durations are naturally handled, provided there are sufficient instances scheduled through the servers to balance the variation.

- *Static scheduling*: if scheduling is statically determined at start-up, then no master processor is needed to direct traffic and a peer partitioning approach is applicable. If the static schedule is a good one (ideal conditions), then this approach will have superior performance. However, heterogeneity, when not known *a priori*, can very quickly degrade performance since there is no mechanism to adapt.

In addition, the following scheduling approach is provided by PICO for the scheduling of concurrent optimizations within the branch and bound strategy:

- *Distributed scheduling*: in this approach, a peer partition is used and each peer maintains a separate queue of pending jobs. When one peer's queue is smaller than the other queues, it requests work from its peers (prior to idleness). In this way, it can adapt to heterogeneous conditions, provided there are sufficient instances to balance the variation. Each partition performs communication between computations, and no processors are dedicated to scheduling. Furthermore, it distributes scheduling load beyond a single processor, which can be important for large numbers of concurrent jobs (whose scheduling might overload a single master) or for fault tolerance (avoiding a single point of failure). However, it involves relatively complicated logic and additional communication for queue status and job migration, and its performance is not always superior since a partition can become work-starved if its peers are locked in computation (Note: this logic can be somewhat simplified if a separate thread can be created for communication and migration of jobs).

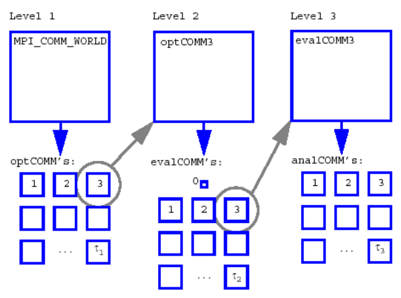


Figure 15.2: Recursive partitioning for nested parallelism.

DAKOTA is designed to allow the freedom to configure each parallelism level with either the dedicated master partition/self-scheduling combination or the peer partition/static scheduling combination. In addition, certain external libraries may provide additional options (e.g., PICO supports distributed scheduling in peer partitions). As an example, Figure 15.2 shows a case in which a branch and bound strategy employs peer partition/distributed scheduling at level 1, each optimizer partition employs concurrent function evaluations in a dedicated master partition/self-scheduling model at level 2, and each function evaluation partition employs concurrent multiprocessor analyses in a peer partition/static scheduling model at level 3. In this case, `MPI_COMM_WORLD` is subdivided into `optCOMM1/2/3/.../`τ_1, each `optCOMM` is further subdivided into `evalCOMM0` (master) and `evalCOMM1/2/3/.../`τ_2 (slaves), and each slave `evalCOMM` is further subdivided into `analCOMM1/2/3/.../`τ_3.

Currently, each message passing scheduler is blocking, in that all jobs in the queue are completed before exiting the scheduler and returning the set of results to the algorithm. Nonblocking message-passing schedulers are under development for the case of fully asynchronous algorithms which do not contain synchronization points (e.g., the APPS algorithm).

15.5 Putting the Components Together

The asynchronous local approaches described in Section 15.3 can be considered to rely on `external` scheduling mechanisms, since it is generally the operating system or some external queue/load sharing software that allocates jobs to processors. Conversely, the message-passing approaches described in Section 15.4 rely on *internal* scheduling mechanisms to distribute work among processors. These components provide building blocks which can be combined in a variety of ways to manage parallelism at multiple levels. At one extreme, DAKOTA can execute on a single processor and rely completely on external means to map all jobs to processors (i.e., using asynchronous local approaches). At the other extreme, DAKOTA can execute on many processors and manage all levels of parallelism, including the parallel simulations, using completely internal approaches (i.e., using message passing at all levels as in Figure 15.2). While all-internal or all-external approaches are common cases, many additional approaches exist between the two extremes in which some parallelism is managed internally and some is managed externally.

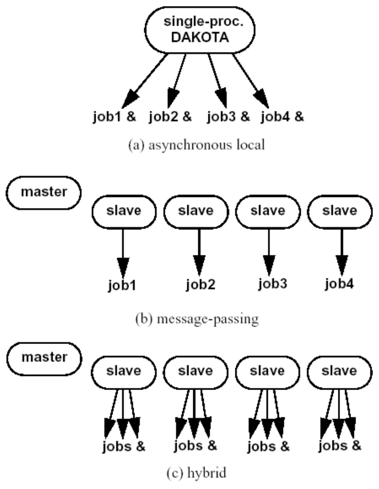


Figure 15.3: External, internal, and hybrid job management.

These combined approaches are referred to as *hybrid* parallelism, since the internal distribution of work based on message-passing is being combined with external allocation using asynchronous local approaches. Figure 15.3 depicts the asynchronous local, message-passing, and hybrid approaches for a dedicated-master partition. Approaches (b) and (c) both use MPI message-passing to distribute work from the master to the slaves, and approaches (a) and (c) both manage asynchronous jobs local to a processor. The hybrid approach (c) can be seen to be a combination of (a) and (b) since jobs are being internally distributed to slave servers through message-passing and each slave server is managing multiple concurrent jobs using an asynchronous local approach. From a different perspective, one could consider (a) and (b) to be special cases within the range of configurations supported by (c). The hybrid approach is useful for supercomputers that maintain a service/compute node distinction and for supercomputers or networks of workstations that involve clusters of symmetric multiprocessors (SMPs). In the service/compute node case, concurrent multiprocessor simulations are launched into the compute nodes from the service node partition. While an asynchronous local approach from a single service node would be sufficient, spreading the application load by running DAKOTA in parallel across multiple service nodes results in better performance [22]. If the number of concurrent jobs to be managed in the compute partition exceeds the number of available service nodes, then hybrid parallelism is the preferred approach. In the case of a cluster of SMPs, message-passing can be used to communicate between SMPs, and asynchronous local approaches can be used within an SMP. Hybrid parallelism can again result in improved performance, since the total number of DAKOTA MPI processes is reduced in comparison to a pure message-passing approach.

Hybrid parallelism approaches can take several forms when used in the multilevel parallel context. A conceptual boundary can be considered to exist for which all parallelism above the boundary is managed internally using message-passing and all parallelism below the boundary is managed externally using asynchronous local approaches. Hybrid parallelism approaches can then be categorized based on whether this boundary between internal and external management occurs within a parallelism level (*intra-level*) or between two parallelism levels (*inter-level*). In the intra-level case, the jobs for the parallelism level containing the boundary are scheduled using a hybrid scheduler, in which a capacity multiplier is used for the number of jobs to assign to each server. Each server is then responsible for concurrently executing its capacity of jobs using an asynchronous local approach. In the inter-level case, one level of parallelism manages its parallelism internally using a message-passing approach and the next lower level of parallelism manages its parallelism externally using an asynchronous local approach. That is, the jobs for the higher level of parallelism are scheduled using a standard message-passing scheduler, in which a single job is as-

Table 15.1: Support of job management approaches within parallelism levels and application interfaces

Parallelism Level	Asynchronous Local	Message Passing	Hybrid
strategy/iterators		**X**	
iterator/function evaluations	**X** (system, fork)	**X** (system, fork, direct)	**X** (system, fork)
function evaluation/analyses	**X** (fork only)	**X** (system, fork, direct)	**X** (fork only)
fine-grained parallel analysis		**X**	

signed to each server. However, each of these jobs has multiple components, as managed by the next lower level of parallelism, and each server is responsible for executing these sub-components concurrently using an asynchronous local approach. For example, consider a multiprocessor DAKOTA run which involves an iterator scheduling a set of concurrent function evaluations across a cluster of SMPs. A hybrid parallelism approach will be applied in which message-passing parallelism is used between SMPs and asynchronous local parallelism is used within each SMP. In the hybrid intra-level case, multiple function evaluations would be scheduled to each SMP, as dictated by the capacity of the SMPs, and each SMP would manage its own set of concurrent function evaluations using an asynchronous local approach. Any lower levels of parallelism would be serialized. In the hybrid inter-level case, the function evaluations would be scheduled one per SMP, and the analysis components within each of these evaluations would be executed concurrently using asynchronous local approaches within the SMP. Thus, the distinction can be viewed as whether the concurrent jobs on each server in Figure 15.3c reflect the same level of parallelism as that being scheduled by the master (intra-level) or one level of parallelism below that being scheduled by the master (inter-level).

Table 15.1 shows a matrix of the supported job management approaches for each of the parallelism levels and each of the application interfaces. The concurrent iterator and multiprocessor analysis parallelism levels can only be managed with message-passing approaches. In the former case, this is due to the fact that a separate process or thread for an iterator is not currently supported. The latter case reflects a finer point on the definition of external parallelism management. While a multiprocessor analysis can most certainly be launched (using `mpirun`/`yod`) from one of DAKOTA's analysis drivers, resulting in a parallel analysis external to DAKOTA, this parallelism is not visible to DAKOTA and therefore does not qualify as parallelism that DAKOTA manages (and therefore is not included in Table 15.1). The concurrent evaluation and analysis levels can be managed either with message-passing, asynchronous local, or hybrid techniques, with the exceptions that the direct interface does not support asynchronous operations (asynchronous local or hybrid) at either of these levels and the system call interface does not support asynchronous operations (asynchronous local or hybrid) at the concurrent analysis level. The direct interface restrictions are present since multithreading in not yet supported and the system call interface restrictions result from the inability to manage concurrent analyses within a nonblocking function evaluation system call.

15.6 Running a Parallel DAKOTA Job

15.6.1 Single-processor execution

The command for running DAKOTA on a single-processor and exploiting asynchronous local parallelism is the same as for running DAKOTA on a single-processor for a serial study, e.g.:

```
dakota -i dakota.in > dakota.out
```

See Section 2.1.5 for additional information on single-processor command syntax.

15.6.2 Multiprocessor execution

Running a DAKOTA job on multiple processors requires the use of an executable loading facility such as mpirun or yod. On a network of workstations, the mpirun script is used to initiate a parallel DAKOTA job, e.g.:

```
mpirun -np 12 dakota -i dakota.in > dakota.out
mpirun -machinefile machines -np 12 dakota -i dakota.in > dakota.out
```

where both examples specify the use of 12 processors, the former selecting them from a default system resources file and the latter specifying particular machines in a machine file (see [38] for details).

On a massively parallel computer such as ASCI Red, similar facilities are available from the Cougar operating system via the yod executable loading facility:

```
yod -sz 512 dakota -i dakota.in > dakota.out
```

In both the mpirun and yod cases, MPI command line arguments are used by MPI (extracted first in the call to MPI_Init) and DAKOTA command line arguments are used by DAKOTA (extracted second by DAKOTA's command line handler). An issue that can arise with these command line arguments is that the mpirun script distributed with MPICH has been observed to have problems with certain file path specifications (e.g., a relative path such as "../some_file"). These path problems are most easily resolved by using local linkage (all referenced files or soft links to these files appear in the same directory).

Finally, when running on computer resources that employ NQS/PBS batch schedulers, the single-processor dakota command syntax or the multiprocessor mpirun command syntax might be contained within an executable script file which is submitted to the batch queue. For example, on Cplant, the command

```
qsub -l size=512 run_dakota
```

could be submitted to the PBS queue for execution. On ASCI Red, the NQS syntax is similar:

```
qsub -q snl -lP 512 -lT 6:00:00 run_dakota
```

These commands allocate 512 compute nodes for the study, and execute the run_dakota script on a service node. If this script contains a single-processor dakota command, then DAKOTA will execute on a single service node from which it can launch parallel simulations into the compute nodes using analysis drivers that contain yod commands (any yod executions occurring at any level underneath the run_dakota script are mapped to the 512 compute node allocation). If the script submitted to qsub contains a multiprocessor mpirun command, then DAKOTA will execute across multiple service nodes so that it can spread the application load in either a message-passing or hybrid parallelism approach. Again, analysis drivers containing yod commands would be responsible for utilizing the 512 compute nodes. And, finally, if the script submitted to qsub contains a yod of the dakota executable, then DAKOTA will execute directly on the compute nodes and manage all of the parallelism internally (note that a yod of this type without a qsub would be mapped to the interactive partition, rather than to the batch partition).

15.7 Specifying Parallelism

Given an allotment of processors, DAKOTA contains logic based on the theoretical work in [22] to automatically determine an efficient parallel configuration, consisting of partitioning and scheduling selections for each of the parallelism levels. This logic accounts for problem size, the concurrency supported by particular iterative algorithms, and any user inputs or overrides. The following points are important components of the automatic configuration logic which can be helpful in estimating the total number of processors to allocate and in selecting configuration overrides:

- If the capacity of the servers in a peer configuration is sufficient to schedule all jobs in one pass, then a peer partition and static schedule will be selected. If this capacity is not sufficient, then a dedicated-master partition and dynamic schedule will be used. These selections can be overridden with self/static scheduling request specifications for the concurrent iterator, evaluation, and analysis parallelism levels. For example, if it is known that processor speeds and job durations have little variability, then overriding the automatic configuration with a static schedule request could eliminate the unnecessary loss of a processor to scheduling.
- With the exception of the concurrent-iterator parallelism level (iterator executions tend to have high variability in duration), concurrency is pushed up. That is, available processors will be assigned to concurrency at the higher parallelism levels first. If more processors are available than needed for concurrency at a level, then the server size is increased to support concurrency in the next lower level of parallelism. This process is continued until all available processors have been assigned. These assignments can be overridden with a servers specification for the concurrent iterator, evaluation, and analysis parallelism levels and with a processors per analysis specification for the multiprocessor analysis parallelism level. For example, if it is desired to parallelize concurrent analyses within each function evaluation, then an `evaluation_servers = 1` override would serialize the concurrent function evaluations level and assure processor availability for concurrent analyses.

In the following sections, the user inputs and overrides are described, followed by specification examples for single and multi-processor DAKOTA executions.

15.7.1 The interface specification

Specifying parallelism within an interface can involve the use of the `asynchronous`, `evaluation_concurrency`, and `analysis_concurrency` keywords to specify concurrency local to a processor (i.e., asynchronous local parallelism). This `asynchronous` specification has dual uses:

- When running DAKOTA on a single-processor, the `asynchronous` keyword specifies the use of asynchronous invocations local to the processor (these jobs then rely on external means to be allocated to other processors). The default behavior is to simultaneously launch all function evaluations available from the iterator as well as all available analyses within each function evaluation. In some cases, the default behavior can overload a machine or violate a usage policy, resulting in the need to limit the number of concurrent jobs using the `evaluation_concurrency` and `analysis_concurrency` specifications.
- When executing DAKOTA across multiple processors and managing jobs with a message-passing scheduler, the `asynchronous` keyword specifies the use of asynchronous invocations local to each server processor, resulting in a hybrid parallelism approach (see Section 15.5). In this case, the default behavior is one job per server, which must be overridden with an `evaluation_concurrency` specification and/or an `analysis_concurrency` specification. When a hybrid parallelism approach is specified, the capacity of the servers (used in the automatic configuration logic) is defined as the number of servers times the number of asynchronous jobs per server.

In addition, `evaluation_servers`, `evaluation_self_scheduling`, and `evaluation_static_scheduling` keywords can be used to override the automatic parallelism configuration for concurrent function evaluations; `analysis_servers`, `analysis_self_scheduling`, and `analysis_static_scheduling` keywords can be used to override the automatic parallelism configuration for concurrent analyses; and the `processors_per_analysis` keyword can be used to override the automatic parallelism configuration for the size of multiprocessor analyses. Each of these keywords appears as part of the interface commands specification in the DAKOTA Reference Manual [17].

15.7.2 The strategy specification

To specify concurrency in iterator executions, the `iterator_servers`, `iterator_self_scheduling`, and `iterator_static_scheduling` keywords are used to override the automatic parallelism configuration. See the strategy commands specification in the DAKOTA Reference Manual [17] for additional information.

15.7.3 Single-processor DAKOTA specification

Specifying a single-processor DAKOTA job that exploits parallelism through asynchronous local approaches (see Figure 15.3a) requires inclusion of the `asynchronous` keyword in the interface specification. Once the input file is defined, single-processor DAKOTA jobs are executed using the command syntax described previously in Section 15.6.1.

Example 1

For example, the following specification runs an NPSOL optimization which will perform asynchronous finite differencing:

```
method,                                                    \
        npsol_sqp

variables,                                                 \
        continuous_design = 5                              \
          cdv_initial_point  0.2  0.05 0.08 0.2  0.2     \
          cdv_lower_bounds   0.15 0.02 0.05 0.1  0.1     \
          cdv_upper_bounds   2.0  2.0  2.0  2.0  2.0

interface,                                                 \
        application system,                                \
          asynchronous                                     \
          analysis_drivers = 'text_book'

responses,                                                 \
        num_objective_functions = 1                        \
        num_nonlinear_inequality_constraints = 2           \
        numerical_gradients                                \
          interval_type central                            \
          method_source dakota                             \
          fd_step_size = 1.0E-4                            \
        no_hessians
```

Note that `method_source dakota` selects DAKOTA's internal finite differencing routine so that the concurrency in finite difference offsets can be exploited. In this case, central differencing has been selected and 11 function evaluations (one at the current point plus two offsets in each of five variables) can be performed simultaneously for each NPSOL response request. These 11 evaluations will be launched with system calls in the background and presumably assigned to additional processors through the operating system of a multiprocessor compute server or other comparable method. The concurrency specification may be included if it is necessary to limit the maximum number of simultaneous evaluations. For example, if a maximum of six compute processors were available, the command

```
evaluation_concurrency = 6                                 \
```

should be added to the `asynchronous` specification in the preceding example.

Example 2

If, in addition, multiple analyses can be executed concurrently within a function evaluation (e.g., from multiple load cases or disciplinary analyses that must be evaluated to compute the response data set), then an input specification similar to the following could be used:

```
method,                                                          \
        npsol_sqp

variables,                                                       \
        continuous_design = 5                                    \
          cdv_initial_point  0.2  0.05 0.08 0.2  0.2             \
          cdv_lower_bounds   0.15 0.02 0.05 0.1  0.1             \
          cdv_upper_bounds   2.0  2.0  2.0  2.0  2.0

interface,                                                       \
        application fork                                         \
          asynchronous                                           \
            evaluation_concurrency = 6                           \
            analysis_concurrency = 3                             \
          analysis_drivers = 'text_book1' 'text_book2' 'text_book3'

responses,                                                       \
        num_objective_functions = 1                              \
        num_nonlinear_inequality_constraints = 2                 \
        numerical_gradients                                      \
          method_source dakota                                   \
          interval_type central                                  \
          fd_step_size = 1.e-4                                   \
        no_hessians
```

In this case, the default concurrency with just an `asynchronous` specification would be all 11 function evaluations and all 3 analyses, which can be limited by the `evaluation_concurrency` and `analysis_concurrency` specifications. The input file above limits the function evaluation concurrency, but not the analysis concurrency (a specification of 3 is the default in this case and could be omitted). Changing the input to `evaluation_concurrency = 1` would serialize the function evaluations, and changing the input to `analysis_concurrency = 1` would serialize the analyses.

15.7.4 Multiprocessor DAKOTA specification

In multiprocessor executions, server evaluations are synchronous (Figure 15.3b) by default and the `asynchronous` keyword is only used if a hybrid parallelism approach (Figure 15.3c) is desired. Multiprocessor DAKOTA jobs are executed using the command syntax described previously in Section 15.6.2.

Example 3

To run Example 1 using a message-passing approach, the `asynchronous` keyword would be removed (since the servers will execute their evaluations synchronously), resulting in the following interface specification:

```
interface,                                          \
        application system,                         \
          analysis_drivers = 'text_book'
```

Running DAKOTA on 4 processors (syntax: `mpirun -np 4 dakota -i dakota.in`) would result in the following parallel configuration report from the DAKOTA output:

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1                4              peer/static
concurrent evaluations       3                1              ded. master/self
concurrent analyses          1                1              peer/static
multiprocessor analysis      1               N/A                N/A

Total parallelism levels =   1
-----------------------------------------------------------------------------
```

The dedicated master partition and self-scheduling algorithm are automatically selected for the concurrent evaluations parallelism level since the number of function evaluations (11) is greater than the maximum capacity of the servers (4). Since one of the processors is dedicated to being the master, only 3 processors are available for computation and the 11 evaluations can be completed in approximately 4 passes through the servers. If it is known that there is little variability in evaluation duration, then this logic could be over-ridden to use a static schedule through use of the `evaluation_static_scheduling` specification:

```
interface,                                                \
        application system,                               \
          evaluation_static_scheduling                    \
          analysis_drivers = ’text_book’
```

Running DAKOTA again on 4 processors (syntax: `mpirun -np 4 dakota -i dakota.in`) would now result in this parallel configuration report:

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1                4              peer/static
concurrent evaluations       4                1              peer/static
concurrent analyses          1                1              peer/static
multiprocessor analysis      1               N/A                N/A

Total parallelism levels =   1
-----------------------------------------------------------------------------
```

Now the 11 jobs will be statically distributed among 4 peer servers, since the processor previously dedicated to scheduling has been converted to a compute server. This will likely be more efficient if the evaluation durations are sufficiently similar.

As a related example, consider the case where each of the workstations used in the parallel execution has multiple processors. In this case, a hybrid parallelism approach which combines message-passing parallelism with asynchronous local parallelism (see Figure 15.3c) would be a good choice. To specify hybrid parallelism, one uses the same `asynchronous` specification as was used for the single-processor examples, e.g.:

```
interface,                                                \
         application system                               \
           asynchronous evaluation_concurrency = 3        \
           analysis_drivers = ‘text_book’
```

With 3 function evaluations concurrent on each server, the capacity of a 4 processor DAKOTA execution (syntax: `mpirun -np 4 dakota -i dakota.in`) has increased to 12 evaluations. Since all 11 jobs can now be scheduled in a single pass, a static schedule is automatically selected (without any override request):

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1                 4              peer/static
concurrent evaluations       4                 1              peer/static
concurrent analyses          1                 1              peer/static
multiprocessor analysis      1                N/A                 N/A

Total parallelism levels =   1
-----------------------------------------------------------------------------
```

Example 4

To run Example 2 using a message-passing approach, the `asynchronous` specification is again removed:

```
interface,                                                             \
         application fork                                              \
           analysis_drivers = 'text_book1' 'text_book2' 'text_book3'
```

Running this example on 6 processors (syntax: `mpirun -np 6 dakota -i dakota.in`) would result in the following parallel configuration report:

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         1                 6              peer/static
concurrent evaluations       5                 1              ded. master/self
concurrent analyses          1                 1              peer/static
multiprocessor analysis      1                N/A                 N/A

Total parallelism levels =   1
-----------------------------------------------------------------------------
```

in which all of the processors have been assigned to the function evaluation concurrency (due to the “push up” automatic configuration logic). To assign some of the available processors to the concurrent analysis level, the following input could be used:

```
interface,                                                             \
         application fork                                              \
           analysis_drivers = 'text_book1' 'text_book2' 'text_book3' \
           evaluation_static_scheduling                                \
           evaluation_servers = 2
```

which results in the following 2-level parallel configuration:

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
```

```
-----                     -----------    ----------------    ------------------
concurrent iterators           1                6              peer/static
concurrent evaluations         2                3              peer/static
concurrent analyses            3                1              peer/static
multiprocessor analysis        1               N/A                N/A

Total parallelism levels =     2
-------------------------------------------------------------------------------
```

The six processors available have been split into two evaluation servers of three processors each, where the three processors in each evaluation server manage the three analyses, one per processor.

Next, consider the following 3-level parallel case, in which `text_book1`, `text_book2`, and `text_book3` from the previous examples now execute on two processors each. In this case, the `processors_per_analysis` keyword is added and the `fork` interface is changed to a `direct` interface since the fine-grained parallelism of the three simulations is managed internally:

```
interface,                                                             \
         application direct                                            \
           analysis_drivers = ‘text_book1’ ‘text_book2’ ‘text_book3’ \
           evaluation_static_scheduling                                \
           evaluation_servers = 2                                      \
           processors_per_analysis = 2
```

This results in the following parallel configuration for a 12 processor DAKOTA run
(syntax: `mpirun -np 12 dakota -i dakota.in`):

```
-------------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                     num_servers    procs_per_server    partition/schedule
-----                     -----------    ----------------    ------------------
concurrent iterators           1               12              peer/static
concurrent evaluations         2                6              peer/static
concurrent analyses            3                2              peer/static
multiprocessor analysis        2               N/A                N/A

Total parallelism levels =     3
-------------------------------------------------------------------------------
```

An important point to recognize is that, since each of the parallel configuration inputs has been tied to the interface specification up to this point, these parallel configurations can be reallocated for each interface in a multi-iterator/multi-model strategy. For example, a DAKOTA execution on 40 processors might involve the following two interface specifications:

```
interface,                                           \
        application direct,                          \
          id_interface = ’COARSE’                    \
          analysis_driver = ’sim1’                   \
          processors_per_analysis = 5

interface,                                           \
        application direct,                          \
          id_interface = ’FINE’                      \
          analysis_driver = ’sim2’                   \
          processors_per_analysis = 10
```

for which the coarse model would employ 8 servers of 5 processors each and the fine model would employ 4 servers of 10 processors each.

Next, consider the following 4-level parallel case that employs the Pareto set optimization strategy. In this case, `iterator_servers` and `iterator_static_scheduling` requests are included in the strategy specification:

```
strategy,                                                \
         pareto_set                                      \
           iterator_servers = 2                          \
           iterator_static_scheduling                    \
           opt_method_pointer = 'NLP'                    \
           random_weight_sets = 4
```

Adding this strategy specification to the input file from the previous 12 processor example results in the following parallel configuration for a 24 processor DAKOTA run
(syntax: `mpirun -np 24 dakota -i dakota.in`):

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         2               12              peer/static
concurrent evaluations       2                6              peer/static
concurrent analyses          3                2              peer/static
multiprocessor analysis      2               N/A                N/A

Total parallelism levels =   4
-----------------------------------------------------------------------------
```

Example 5

As a final example, consider a multi-start optimization conducted on 384 processors of ASCI Red. A job of this size must be submitted to the batch queue, using syntax similar to:

```
qsub -q snl -lP 384 -lT 6:00:00 run_dakota
```

where the `run_dakota` script appears as

```
#!/bin/sh
date
cd /scratch/tmp_5/mseldre
yod -sz 384 dakota -i dakota.in > dakota.out
date
```

and the strategy and interface specifications from the `dakota.in` input file appear as

```
strategy,                                                           \
        multi_start                                                 \
          method_pointer = 'CPS'                                    \
          iterator_servers = 8                                      \
          random_starts = 8

interface,                                                          \
        application direct,                                         \
          analysis_drivers = 'text_book1' 'text_book2' 'text_book3' \
          evaluation_servers = 8                                    \
          evaluation_static_scheduling                              \
          processors_per_analysis = 2
```

The resulting parallel configuration is reported as

```
-----------------------------------------------------------------------------
DAKOTA parallel configuration:

Level                   num_servers    procs_per_server    partition/schedule
-----                   -----------    ----------------    ------------------
concurrent iterators         8               48                peer/static
concurrent evaluations       8                6                peer/static
concurrent analyses          3                2                peer/static
multiprocessor analysis      2               N/A                  N/A

Total parallelism levels =   4
-----------------------------------------------------------------------------
```

Since the concurrency at each of the nested levels has a multiplicative effect on the number of processors that can be utilized, it is easy to see how large numbers of processors can be put to effective use in reducing the time to reach a solution, even when, as in this example, the concurrency per level is relatively low.

Chapter 16

Advanced Simulation Code Interfaces

16.1 Building an Interface to a Engineering Simulation Code

To interface an engineering simulation package to DAKOTA using one of the black-box interfaces (system call or fork), pre- and post-processing functionality typically needs to be supplied (or developed) in order to transfer the parameters from DAKOTA to the simulator input file and to extract the response values of interest from the simulator's output file for return to DAKOTA (see Figure 16.1). This is often managed through the use of a UNIX C-shell [1], Bourne shell [7], or Perl [68] driver script. While these are common choices, it is important to recognize that any executable file can be used. If the user prefers, the desired pre- and post-processing functionality may also be provided by an executable compiled from any programming language.

Under the `/Dakota/GettingStarted/RosenSimulator` directory, a simple example uses the Rosenbrock test function as a mock simulator. Several scripts have been included to provide ways to accomplish the pre and post-processing needs. Each simulator package has different pre- and post-processing requirements, and as such, this example serves only to demonstrate the issues associated with interfacing a simulator. Modifications will almost surely be required for any particular application.

16.1.1 Review of RosenSimulator Files

The RosenSimulator directory contains four important files: `dakota_rosenbrock.in` (the DAKOTA input file), `simulator_script` (the simulation driver script), `dprepro` (a pre-processing utility), and `templatedir/ros.template` (a template simulation input file).

The `dakota_rosenbrock.in` file specifies the study that DAKOTA will perform and, in the interface section, describes the components to be used in performing function evaluations. In particular, it identifies `simulator_script` as its `analysis_driver`, as shown in Figure 16.1.

The `simulator_script` listed in Figure 16.2 is a short C-shell driver script that DAKOTA executes to perform each function evaluation. The names of the parameters and results files are passed to the script on its command line so that they can be referenced internal to the script by the variables `$argv[1]` and `$argv[2]`, respectively. The `simulator_script` is divided into five parts: set up, pre-processing, analysis, post-processing, and clean up.

The set up portion strips the function evaluation number from `$argv[1]`and assigns it to the shell variable `$num`, which is then used to create a tagged working directory for a particular evaluation. For example, on the first evaluation, "`1`" is stripped from "`params.in.1`" in order to create "`workdir.1`". The primary reason for creating separate working directories is so that the files associated with one simulation do not conflict with those for another simulation. This is particularly important when executing concurrent

```
#  DAKOTA INPUT FILE - dakota_rosenbrock.in
#  This sample Dakota input file optimizes the Rosenbrock function.
#  See p. 95 in Practical Optimization by Gill, Murray, and Wright.
#

method,                                         \
        npsol_sqp

variables,                                      \
        continuous_design = 2                   \
        cdv_initial_point   -1.0      1.0       \
        cdv_lower_bounds    -2.0     -2.0       \
        cdv_upper_bounds     2.0      2.0       \
        cdv_descriptor       'x1'     'x2'

interface,                                      \
        system,                                 \
#       asynchronous                            \
          analysis_driver = 'simulator_script'  \
          parameters_file = 'params.in'         \
          results_file    = 'results.out'       \
          file_tag                              \
          file_save                             \
          aprepro

responses,                                      \
        num_objective_functions = 1             \
        numerical_gradients                     \
          fd_gradient_step_size = .000001       \
        no_hessians
```

Figure 16.1: The `dakota_rosenbrock.in` input file.

```
#!/bin/csh -f
# Sample simulator to Dakota system call script
# See User Manual for instructions
#
# bvbw 10/24/01
#  slb 09/20/05

# $argv[1] is params.in.(fn_eval_num) FROM Dakota
# $argv[2] is results.out.(fn_eval_num) returned to Dakota

# ------------------------
# Set up working directory
# ------------------------

set num = ‘echo $argv[1] | cut -c 11-‘

cp -r templatedir workdir.$num
mv $argv[1] workdir.$num
cd workdir.$num

# --------------
# PRE-PROCESSING
# --------------
# Use the following line if SNL’s APREPRO utility is used instead
# of dprepro.
# ../aprepro -c ’*’ -q --nowarning ros.template ros.in

../dprepro $argv[1] ros.template ros.in

# --------
# ANALYSIS
# --------

../rosenbrock_bb

# ---------------
# POST-PROCESSING
# ---------------

grep ’Function value’ ros.out | cut -c 18- >! $argv[2]
# NOTE: moving $argv[2] at the end of the script avoids any
# problems with read race conditions.
mv $argv[2] ../.

# --------
# Clean up
# --------

cd ..
\rm -rf workdir.$num
```

Figure 16.2: The `simulator_script` sample driver script.

simulations in parallel (to actually run DAKOTA in parallel, uncomment the `asynchronous` line in `dakota_rosenbrock.in`). Once executing within the confines of the working directory, tags on the files are no longer necessary, and for this reason, the tagged parameters file is moved to a more convenient name of "`dakota_vars`".

In the pre-processing portion, the `simulator_script` utilizes `dprepro`, which is a parsing utility used to extract the current variable values from a parameters file (`dakota_vars`) and then insert them into the simulator template input file (`ros.template`) to create a new input file (`ros.in`) for the simulator. Internal to Sandia, the APREPRO utility is often used for this purpose. For external sites where APREPRO is not available, the DPREPRO utility mentioned above is an alternative with many of the capabilities of APREPRO that is distributed with DAKOTA (in `/Dakota/GettingStarted/RosenSimulator/dprepro`). Additionally, the BPREPRO utility is a capable alternative to APREPRO (see [69]), and at Lockheed Martin sites, the JPrepro utility is available as a JAVA pre- and post-processor [26]. The `dprepro` script partially listed in Figure 16.3 provides a pre-processing capability and will be used here for simplicity of discussion. It can use either DAKOTA's `aprepro` parameters file format (see Section 4.6.2) or DAKOTA's standard format ("`value tag`"), so one of these options must be selected in the interface section of the DAKOTA input file. The `ros.template` file listed in Figure 16.4 is a template simulation input file which contains targets for the incoming variable values, identified by the strings "{`x1`}" and "{`x2`}". These identifiers match the variable descriptors specified in `dakota_rosenbrock.in`. The template input file is contrived as Rosenbrock has nothing to do with finite element analysis; it only mimics a finite element code in order to demonstrate the simulator template process. The `dprepro` script will search the simulator template input file for fields marked with the curly brackets and then create a new file (`ros.in`) by replacing these targets with the corresponding numerical values for the variables. As noted in the usage information for `dprepro`, the DAKOTA parameters file ("`dakota_vars`"), template file name ("`ros.template`") and generated input file ("`ros.in`") must be specified in the arguments for it.

The third part of the script executes the `rosenbrock_bb` simulator. The input and output file names, `ros.in` and `ros.out`, respectively, are hard-coded into the FORTRAN 77 program `rosenbrock_bb.f`. When the `rosenbrock_bb` simulator is executed, the values for `x1` and `x2` are read in from `ros.in`, the Rosenbrock function is evaluated, and the function value is written out to `ros.out`.

The fourth part performs the post-processing and returns the response results to DAKOTA. Using the UNIX "`grep`" utility, the particular response values of interest are extracted from the raw simulator output and saved to `$argv[2]`, which in the case of the first evaluation is "`results.out.1`". This results file is moved up one level, out of the working directory, so that DAKOTA may retrieve it. Note that moving the completed results file up a level at the end of the evaluation avoids any problems with read race conditions (see Section 15.3.2).

Finally, in the clean up phase, the working directory is removed to reduce the amount of disk storage required to execute the study. If data from each simulation needs to be saved, this step can be commented out by inserting a "#" character before "`\rm -rf`".

As an example, consider function evaluation 60. The `dakota_vars` file for this evaluation consists of:

```
{ DAKOTA_VARS     = 2 }
{ x1              =  1.6379575982e-01 }
{ x2              =  2.1962319919e-02 }
{ DAKOTA_FNS      = 1 }
{ ASV_1           =                 1 }
{ DAKOTA_DER_VARS = 2 }
{ DVV_1           =                 1 }
{ DVV_2           =                 2 }
{ DAKOTA_AN_COMPS = 0 }
```

This file indicates that there are two variables and one response function (an objective function) and pro-

```
#!/usr/bin/perl
#
# DPREPRO: A Perl pre-processor for manipulating input files with DAKOTA.
# ______________________________________________________________________
#
# Copyright (c) 2001, Sandia National Laboratories.
# This software is distributed with DAKOTA under the GNU GPL.
# For more information, see the README file in the top Dakota directory.
#
# Usage: dprepro parameters_file template_input_file new_input_file
#
# Reads the variable tags and values from the parameters_file and then
# replaces each appearance of "{tag}" in the template_input_file with
# its associated value in order to create the new_input_file.  The
# parameters_file written by DAKOTA may either be in standard format
# (using "value tag" constructs) or in "aprepro" format (using
# "{ tag = value }" constructs), and the variable tags used inside
# template_input_file must match the variable descriptors specified in
# the DAKOTA input file.  Supports assignments and numerical expressions
# in the template file, and the parameters file takes precedence in
# the case of duplicate assignments (so that template file assignments
# can be treated as defaults to be overridden).
# ______________________________________________________________________

# Check for correct number of command line arguments and store the filenames.
if( @ARGV != 3 ) {
  print STDERR "Usage: dprepro parameters_file template_input_file ",
    "new_input_file\n";
  exit(-1);
}
$params_file   = $ARGV[0]; # DAKOTA parameters file in APREPRO format
$template_file = $ARGV[1]; # template simulation input file
$new_file      = $ARGV[2]; # new simulation input file with insertions

# Regular expressions for numeric fields
$e  = "-?(?:\\d+\\.?\\d*|\\.\\d+)[eEdD](?:\\+|-)?\\d+"; # exponential notation
$f  = "-?\\d+\\.\\d*|-?\\.\\d+";                        # floating point
$i  = "-?\\d+";                                         # integer
$ui = "\\d+";                                           # unsigned integer
$n  = "$e|$f|$i";                                       # numeric field

################################
# Process DAKOTA parameters file
################################

# Open parameters file for input.
open (DAKOTA_PARAMS, "<$params_file") || die "Can’t open $params_file: $!";
```

Figure 16.3: Partial listing of the dprepro script.

```
Title of Model: Rosenbrock black box
***************************************************************************
* Description:  This is an input file to the Rosenbrock black box
*               Fortran simulator.  This simulator is structured so
*               as to resemble the input/output from an engineering
*               simulation code, even though Rosenbrock's function
*               is a simple analytic function.  The node, element,
*               and material blocks are dummy inputs.
*
* Input:  x1 and x2
* Output: objective function value
***************************************************************************
node 1 location 0.0 0.0
node 2 location 0.0 1.0
node 3 location 1.0 0.0
node 4 location 1.0 1.0
node 5 location 2.0 0.0
node 6 location 2.0 1.0
node 7 location 3.0 0.0
node 8 location 3.0 1.0
element 1 nodes 1 3 4 2
element 2 nodes 3 5 6 4
element 3 nodes 5 7 8 6
element 4 nodes 7 9 10 8
material 1 elements 1 2
material 2 elements 3 4
variable 1 {x1}
variable 2 {x2}
end
```

Figure 16.4: Listing of the `ros.template` file

vides new values for variables x1 and x2 and an active set vector (ASV) with a single value of 1. The ASV indicates the need to return the value of the objective function for these parameters (see Section 4.7).

The dprepro script reads the variable values from the dakota_vars file, namely 1.6379575982e-01 and 2.1962319919e-02 for x1 and x2 respectively, and substitutes them in the {x1} and {x2} fields of the ros.template file. The final three lines of the resulting input file (ros.in) then appear as follows:

```
variable 1 1.6379575982e-01
variable 2 2.1962319919e-02
end
```

where all other lines are identical to the template file. The rosenbrock_bb simulator accepts ros.in as its input file and generates the following output to the file ros.out:

```
Beginning execution of model: Rosenbrock black box
Set up complete.
Reading nodes.
Reading elements.
Reading materials.
Checking connectivity...OK
*****************************************************

Input value for x1 = 0.1637957598200000E+00
Input value for x2 = 0.2196231991900000E-01

Computing solution...Done
*****************************************************
Function value = .7016060383732316552lE+00
```

It is the user's responsibility to extract the appropriate data from the raw simulator output and return the desired data set to the results file. This step is relatively trivial in this case, and we use the grep and cut utilities to extract the value from the last line of the ros.out output file and save it to $argv[2], which is the results.out.60 file for this evaluation. This single value provides the objective function value requested by the ASV.

Figure 16.5 shows the final solution from DAKOTA using the rosenbrock_bb simulator.

16.1.2 Adapting These Scripts to Another Simulation

To adapt this approach for use with another simulator, several steps need to be performed:

1. Create a template simulator input file by identifying the fields in an existing input file that correspond to the variables of interest and then replacing them with {} identifiers (e.g. {var1}, {var2}, etc.) which match the DAKOTA variable descriptors. Copy this template input file to a templatedir that will be used to create working directories for the simulation.

2. Modify the dprepro arguments in simulator_script to reflect names of the DAKOTA parameters file (previously "dakota_vars"), template file name (previously "ros.template") and generated input file (previously "ros.in"). Alternatively, use APREPRO, BPREPRO or JPrepro to perform this step.

3. Modify the analysis section of simulator_script to replace the rosenbrock_bb function call with the new simulator name and command line syntax, including the input and output file names.

```
    Exit NPSOL - Optimal solution found.

    Final nonlinear objective value =   0.1177903E-06

   NPSOL exits with INFORM code = 0 (see "Interpretation of output" section
                                     in NPSOL manual)

   NOTE: see Fortran device 9 file (fort.9 or ftn09)
         for complete NPSOL iteration history.

   <<<<< Iterator npsol_sqp completed.
   <<<<< Function evaluation summary: 132 total (132 new, 0 duplicate)
   <<<<< Best parameters          =
                         9.9965683296e-01 x1
                         9.9931326685e-01 x2
   <<<<< Best objective function  =
                         1.1779032904e-07
   <<<<< Best data captured at function evaluation 130
   <<<<< Single Method Strategy completed.
   DAKOTA execution time in seconds:
     Total CPU        =      0.26 [parent =      0.26, child =         0]
     Total wall clock =    38.044
```

Figure 16.5: DAKOTA output for RosenSimulator problem.

4. Change the post-processing section in `simulator_script` to reflect the revised extraction process. At a minimum, this would involve changing the `grep` command to reflect the name of the output file, the string to search for, and the characters to cut out of the captured output line. For more involved post-processing tasks, invocation of additional tools may have to be added to the script.

5. Modify the `dakota_rosenbrock.in` input file to reflect, at a minimum, the initial values, bounds, and tags in the variables specification and the number of objectives and constraints in the responses specification.

These nonintrusive interfacing approaches can be used to rapidly interface with simulation codes. While generally custom for each new application, typical interface development time is on the order of an hour or two. Thus, this approach is scalable when dealing with many different application codes. Weaknesses of this approach include the potential for loss of data precision (if care is not taken to preserve precision in pre- and post-processing file I/O), a lack of robustness in post-processing (if the data capture is too simplistic), and scripting overhead (only noticeable if the simulation time is on the order of a second or less).

If the application scope at a particular site is more focused and only a small number of simulation codes are of interest, then more sophisticated interfaces may be warranted. For example, the economy of scale afforded by a common simulation framework justifies additional effort in the development of a high quality DAKOTA interface. In these cases, more sophisticated interfacing approaches could involve a more thoroughly developed black box interface with robust support of a variety of inputs and outputs, or it might involve intrusive interfaces such as the direct application interface discussed in Section 16.2 and the SAND interface to be available in future releases.

16.1.3 Additional Examples

A variety of additional examples of black-box interfaces to simulation codes are maintained in the `/Dakota/Applications` directory in the source code distribution.

16.2 Adding Simulations to the Direct Application Interface

If a high performing interface to a simulation is desired or the computer architecture cannot accommodate separate optimization and simulation processes (e.g., due to batch submission requirements on large parallel computers), the simulation code can be directly linked into DAKOTA as a subroutine. This is an advanced capability of DAKOTA, and it requires a user to have access to (and knowledge of) the DAKOTA source code, as well as the source code of the simulation code.

In order to use the direct function capability with a new simulation (or new internal test function), the following steps have to be performed:

1. The functions to be invoked (analysis programs, input and output filters) must have their main programs changed into callable functions/subroutines. If it is practical to introduce a dependence on DAKOTA data types, then these functions/subroutines can directly use the following prototype:

   ```
   int function_name(const DakotaVariables& vars,
        const DakotaIntArray& asv, DakotaResponse& response)
   ```

   If it is desired to avoid this data dependence, then a wrapper function with the same prototype should be added to the DirectFnApplicInterface class in order to provide data mappings (see Salinas and ModelCenter wrappers as examples).

2. The if-else blocks in the **derived_map_if()**, **derived_map_ac()**, and **derived_map_of()** member functions of the **DirectFnApplicInterface class** must be extended to include the new function names with the proper prototypes. If the new function names are not members of the **DirectFnApplicInterface class**, then an `extern` declaration may additionally be needed.

3. The DAKOTA system must be recompiled and linked with the new function object files or libraries.

Various header files may have to be included, particularly within the **DirectFnApplicInterface** class, in order to recognize new external functions and compile successfully. Refer to the DAKOTA Developers Manual [18] for additional information on the **DirectFnApplicInterface** class and the DAKOTA data types.

Chapter 17

DAKOTA Usage Guidelines

17.1 Problem Exploration

The first objective in an analysis is to characterize the problem so that appropriate algorithms can be chosen. In the case of optimization, typical questions that should be addressed include: Are the design variables continuous, discrete, or mixed? Is the problem constrained or unconstrained? How expensive are the response functions to evaluate? Will the response functions behave smoothly as the design variables change or will there be nonsmoothness and/or discontinuities? Are the response functions likely to be multimodal, such that global optimization may be warranted? Is analytic gradient data available, and if not, can I calculate gradients accurately and cheaply? Additional questions that are pertinent for characterization of uncertainty quantification problems include: Can I accurately model the probabilistic distributions of my uncertain variables? Are the response functions relatively linear? Am I interested in a full random process characterization of the response functions, or just statistical results?

If there is not sufficient information from the problem description to answer these questions, then additional problem characterization activities may be warranted. One particularly useful characterization activity that DAKOTA enables is parameter space exploration through the use of parameter studies and design of experiments methods. The parameter space can be systematically interrogated to create sufficient information to evaluate the trends in the response functions and to determine if these trends are noisy or smooth, unimodal or multimodal, relatively linear or highly nonlinear, etc. In addition, the parameter studies may reveal that one or more of the parameters do not significantly affect the results and can be removed from the problem formulation. This can yield a potentially large savings in computational expense for the subsequent studies. Refer to Chapter 8 and Chapter 9 for additional information on parameter studies and design of experiments methods.

17.2 Optimization Method Selection

In selecting an optimization method, important considerations include the type of variables in the problem (continuous, discrete, mixed), whether a global search is needed or a local search is sufficient, and the required constraint support (unconstrained, bound constrained, nonlinearly constrained). Less obvious, but equally important, considerations include the efficiency of convergence to an optimum (i.e., convergence rate) and the robustness of the method in the presence of challenging design space features (e.g., nonsmoothness).

Gradient-based optimization methods are highly efficient, with the best convergence rates of all of the optimization methods. If analytic gradient and Hessian information can be provided by an application code, a full Newton method will provide quadratic convergence rates near the solution. More commonly, only gradient information is available and a quasi-Newton method is chosen in which the Hessian information is approximated from an accumulation of gradient data. In this case, superlinear convergence rates can be obtained. These characteristics make gradient-based optimization methods the methods of choice when the problem is smooth and well-behaved. However, when the problem exhibits nonsmooth, discontinuous, or multimodal behavior, these methods can also be the least robust since inaccurate gradients will lead to bad search directions, failed line searches, and early termination.

Thus, for gradient-based optimization, a critical factor is the gradient accuracy. Analytic gradients are ideal, but are rarely available. For many engineering applications, a finite difference method will be used by the optimization algorithm to estimate gradient values. DAKOTA allows the user to select the step size for these calculations, as well as a choice between forward-difference and central-difference algorithms. The finite difference step size should be selected as small as possible, to allow for local accuracy and convergence, but not so small that the steps are "in the noise." This requires an assessment of the local smoothness of the response functions using, for example, a parameter study method. Central-differencing, in general, will produce more reliable gradients than forward differencing, but at twice the expense.

Nongradient-based methods exhibit much slower convergence rates for finding an optimum, and as a result, tend to be much more computationally demanding than gradient-based methods. Nongradient local optimization methods, such as pattern search algorithms, often require from several hundred to a thousand or more function evaluations, depending on the number of variables, and nongradient global optimization methods such as genetic algorithms may require from thousands to tens-of-thousands of function evaluations. Clearly, for nongradient optimization studies, the computational cost of the function evaluation must be relatively small in order to obtain an optimal solution in a reasonable amount of time. In addition, nonlinear constraint support in nongradient methods is an open area of research and is not yet available in DAKOTA. However, nongradient methods can be more robust and more inherently parallel than gradient-based approaches. They can be applied in situations were gradient calculations are too expensive or unreliable. In addition, some nongradient-based methods can be used for global optimization which gradient-based techniques, by themselves, cannot. For these reasons, nongradient-based methods deserve consideration when the problem may be nonsmooth or poorly behaved.

An approach which attempts to bring the efficiency of gradient-based optimization methods to nonsmooth or poorly behaved problems is the surrogate-based optimization (SBO) strategy. This technique can smooth noisy or discontinuous response results through use of a data fit surrogate model (e.g., a quadratic polynomial) and then optimize on the smooth surrogate using efficient gradient-based techniques. Section 13.7 provides further information on this approach. In addition, the multilevel hybrid and multistart optimization strategies can address a similar goal of bringing the efficiency of gradient-based optimization methods to global optimization problems. In the former case, a global optimization method can be used for a few cycles to locate promising regions and then local gradient-based optimization is used to efficiently converge on one or more optima. In the latter case, a stratification technique is used to disperse a series of local gradient-based optimization runs through parameter space. Section 13.2 and Section 13.3 provide more information on these approaches.

Table 17.1 provides a convenient reference for choosing an optimization method or strategy to match the characteristics of the user's problem. With respect to constraint support, it should be understood that the methods with more advanced constraint support are also applicable to the lower constraint support levels; they are listed only at their highest level of constraint support for brevity.

17.3 UQ Method Selection

The need for computationally efficient methods is further amplified in the case of the quantification of uncertainty in computational simulations. Sampling-based methods are the most robust uncertainty techniques available, are applicable to almost all simulations, and possess rigorous error bounds; consequently, they should be used whenever the function is relatively inexpensive to compute. However, in the case of terascale computational simulations, the number of function evaluations required by traditional techniques

Table 17.1: Guidelines for optimization method selection.

Variable Type	**Function Surface**	**Solution Type**	**Constraints**	**Applicable Methods**
continuous	smooth	local	unconstrained	optpp_cg
			bound constrained	dot_bfgs, dot_frcg, conmin_frcg
			nonlinearly constrained	npsol_sqp, reduced_sqp, dot_mmfd, dot_slp, dot_sqp, conmin_mfd, optpp_newton, optpp_q_newton, optpp_fd_newton
		local least squares	nonlinearly constrained	nlssol_sqp, optpp_g_newton
		local large-scale	nonlinearly constrained	(planned: reduced_sqp for SAND)
		global	nonlinearly constrained	multi_level opt strategy multi_start opt strategy
	nonsmooth	local	unconstrained	coliny_solis_wets
			bound constrained	coliny_pattern_search, coliny_apps, optpp_pds
			nonlinearly constrained	(planned: coliny_apps, coliny_pattern_search)
		local/global	nonlinearly constrained	surrogate_based_opt strategy
		global	bound constrained	coliny_ea
			nonlinearly constrained	(planned: coliny_ea)
discrete	n/a	global	bound constrained	coliny_ea
mixed	smooth	local	nonlinearly constrained	branch_and_bound opt strategy (noncategorical variables only)
	nonsmooth	global	bound constrained	coliny_ea

Table 17.2: Guidelines for UQ method selection.

Method Classification	**Desired Problem Characteristics**	**Applicable Methods**
Sampling	response functions are relatively inexpensive	nond_sampling (Monte Carlo or LHS)
Analytic reliability	scalar response function that is reasonably well behaved	nond_analytic_reliability (MV, AMV, AMV+, FORM, SORM)
Stochastic finite elements	representation of full random variable/process/ field is desired	nond_polynomial_chaos

such as Monte Carlo and Latin hypercube sampling (LHS) quickly becomes prohibitive. One way to alleviate this problem is to employ more advanced sampling strategies, such as Quasi-Monte Carlo (QMC) sampling, importance sampling (IS), or Markov Chain Monte Carlo (MCMC) sampling, and these techniques are currently under investigation.

Alternatively, one can apply the traditional sampling techniques to a surrogate function approximating the expensive computational simulation. However, if this approach is selected, the user should be aware that it is very difficult to assess the accuracy of the results obtained. Unlike in the case of SBO (see Section 13.7), there is no simple pointwise calculation to verify the accuracy of the approximate results. This is due to the functional nature of uncertainty quantification, i.e. the accuracy of the surrogate over the entire parameter space needs to be considered not just around a candidate optimum as in the case of SBO. This issue especially manifests itself when trying to estimate low probability events such as catastrophic failure of a system.

Another class of methods, the analytical reliability methods (MV, AMV, AMV+, FORM/SORM), are more computationally efficient in general than the sampling methods and are effective when applied to reasonably well-behaved response functions, such as linear, mildly nonlinear, and monotonic functions. The user should be aware that these methods do not possess rigorous error estimates as they also involve response surface approximations. Also, they are usually applied only in the scalar response case. Finally, since they rely on gradient calculations, issues with nonsmoothness and poorly behaved response functions are relevant concerns. However, in the case of a small number of uncertain variables, the methods can often be used to provide qualitative sensitivity information concerning which uncertain variables are important with relatively few function evaluations.

The final class of UQ methods available in DAKOTA are stochastic finite elements techniques using polynomial chaos expansions, which are general purpose techniques provided that the response functions possess finite second order moments. Further, these methods approximate the full random process/field rather than just approximating statistics such as mean and standard deviation. This class of methods parallels traditional variational methods in mechanics; in that vein, efforts are underway to compute rigorous error bounds of the approximations produced by the methods. Another strength of the these methods is their potential use in a multiphysics environment as a means to propagate the uncertainty through a series of simulations while retaining as much information as possible at each stage of the analysis. On the other hand, these methods currently rely on the use of traditional sampling techniques in the construction of the approximations; consequently, they are computational very expensive in the case of terascale applications.

The recommendations for UQ methods are summarized in Table 17.2.

Table 17.3: Guidelines for parameter study and design of experiments method selection.

Method Classification	Applications	Applicable Methods
parameter study	sensitivity analysis, directed parameter space investigations	centered_parameter_study, list_parameter_study, multidim_parameter_study, vector_parameter_study
design of computer experiments	main effects analysis, space filling designs (parameters are uniformly distributed)	dace (grid, random, oas, lhs, oa_lhs, box_behnken, central_composite)
sampling	space filling designs (parameters have general probability distributions)	nond_sampling (Monte Carlo or LHS) with all_variables flag

17.4 Parameter Study/DACE/Sampling Method Selection

Parameter studies, design/analysis of computer experiments (DACE), and sampling methods share the purpose of exploring the parameter space. If directed studies with a defined structure are desired, then parameter study methods (see Chapter 8) are recommended. For example, a quick assessment of the smoothness of a response function is best addressed with a vector parameter study. Also, performing local sensitivity analysis is best addressed with these methods. If, however, a global space-filling set of samples is desired, then the DACE and sampling methods are recommended (see Chapter 9). These techniques are useful for scatter plot and main effects analysis as well as surrogate model construction. The distinction between DACE and sampling is drawn based on the distributions of the parameters. Design of experiments methods typically assume uniform distributions, whereas the sampling approaches in DAKOTA support a broad range of probability distributions. To use `nond_sampling` in a design of experiments mode (as opposed to an uncertainty quantification mode), the `all_variables` flag should be included in the method specification of the DAKOTA input file.

These method selection recommendations are summarized in Table 17.3.

Chapter 18

Restart Capabilities and Utilities

18.1 Restart Management

DAKOTA was developed for solving problems that require multiple calls to computationally expensive simulation codes. In some cases you may want to conduct the same optimization, but to a finer final convergence tolerance. This would be costly if the entire optimization analysis had to be repeated. Interruptions imposed by computer usage policies, power outages, and system failures could also result in costly delays. However, DAKOTA automatically records the variable and response data from all function evaluations so that new executions of DAKOTA can pick up where previous executions left off.

The DAKOTA restart file (e.g., `dakota.rst`) is written in a portable binary format. The portability derives from use of the XDR standard.

To write a restart file using a particular name, the `-write_restart` command line input (may be abbreviated as `-w`) is used:

```
    dakota -i dakota.in -write_restart my_restart_file
```

If no `-write_restart` specification is used, then DAKOTA will write a restart file using the default name `dakota.rst`.

To restart DAKOTA from a restart file, the `-read_restart` command line input (may be abbreviated as `-r`) is used:

```
    dakota -i dakota.in -read_restart my_restart_file
```

If no `-read_restart` specification is used, then DAKOTA will not read restart information from any file (i.e., the default is no restart processing).

If the `-write_restart` and `-read_restart` specifications identify the same file (including the case where `-write_restart` is not specified and `-read_restart` identifies `dakota.rst`), then new evaluations will be appended to the existing restart file. If the `-write_restart` and `-read_restart` specifications identify different files, then the evaluations read from the file identified by `-read_restart` are first written to the `-write_restart` file. Any new evaluations are then appended to the `-write_restart` file. In this way, restart operations can be chained together indefinitely with the assurance that all of the relevant evaluations are present in the latest restart file.

To read in only a portion of a restart file, the `-stop_restart` control (may be abbreviated as `-s`) is used. Note that the integer value specified refers to the number of entries to be read from the database, which may differ from the evaluation number in the previous run (e.g., if any duplicates were detected since these duplicates are not recorded in the restart file). In the case of a `-stop_restart` specification, it is usually desirable to specify a new restart file using `-write_restart` so as to remove the records of erroneous or corrupted function evaluations. For example, to read in the first 50 evaluations from `dakota.rst`:

```
    dakota -i dakota.in -read_restart dakota.rst \\
         -stop_restart 50 -write_restart dakota_new.rst
```

The `dakota_new.rst` file will contain the 50 processed evaluations from `dakota.rst` as well as any new evaluations. All evaluations following the 50^{th} in `dakota.rst` have been removed from the latest restart record.

DAKOTA's restart algorithm relies on its duplicate detection capabilities. Processing a restart file populates the list of function evaluations that have been performed. Then, when the study is restarted, it is started from the beginning (not a “warm” start) and many of the function evaluations requested by the iterator are intercepted by the duplicate detection code. This approach has the primary advantage of restoring the complete state of the iteration (including the ability to correctly detect subsequent duplicates) for all iterators and multi-iterator strategies without the need for iterator-specific restart code. However, the possibility exists for numerical round-off error to cause a divergence between the evaluations performed in the previous and restarted studies. This has been extremely rare to date.

```
Usage:  "dakota_restart_util to_neutral <restart_file> <neutral_file>"
        "dakota_restart_util from_neutral <neutral_file> <restart_file>"
        "dakota_restart_util to_pdb <restart_file> <pdb_file>"
        "dakota_restart_util to_tabular <restart_file> <text_file>"
        "dakota_restart_util remove <double> <old_restart_file> <new_restart_file>"
        "dakota_restart_util remove_ids <int_1> ... <int_n> <old_restart_file>
        <new_restart_file>"
        "dakota_restart_util cat <restart_file_1> ... <restart_file_n>
        <new_restart_file>"
```

18.2 The DAKOTA Restart Utility

The DAKOTA restart utility program provides a variety of facilities for managing restart files from DAKOTA executions. The executable program name is `dakota_restart_util` and it has the following options, as shown by the usage message returned when executing the utility without any options:

Several of these functions involve format conversions. In particular, the binary format used for restart files can be converted to ASCII text and printed to the screen, converted to and from a neutral file format, converted to a PDB format for use at Lawrence Livermore National Laboratory, or converted to a tabular format for importing into 3rd-party graphics programs. In addition, a restart file with corrupted data can be repaired by value or id and multiple restart files can be combined to create a master database.

18.2.1 Print

The `print` option is quite useful for interrogating the contents of a particular restart file, since the binary format is not convenient for direct inspection. The restart data is printed in full precision, so that exact matching of points is possible for restarted runs or corrupted data removals. For example, the following command

```
    dakota_restart_util print dakota.rst
```

results in output similar to the following (from the `cyl_head` example problem):

```
    ------------------------------------------
```

```
Function evaluation    1 from restart file:
-------------------------------------------
                      1.8000000000000000e+00 intake_dia
                      1.0000000000000000e+00 flatness
Active set vector = { 1 1 1 1 }
                     -2.4355973813000000e+00 f1
                     -4.7428486676999998e-01 f2
                     -4.5000000000000001e-01 f3
                      1.3971143170000000e-01 f4

-------------------------------------------
Function evaluation    2 from restart file:
-------------------------------------------
                      1.8001800000000001e+00 intake_dia
                      1.0000000000000000e+00 flatness
Active set vector = { 1 1 1 1 }
                     -2.4356759411000000e+00 f1
                     -4.7425991059000000e-01 f2
                     -4.5000000000000001e-01 f3
                      1.3971143170000000e-01 f4
... <<omission>> ...
```

All function evaluations will be printed to the screen, so piping this output into more, e.g.

```
dakota_restart_util print dakota.rst | more
```

or redirecting the output to a file, e.g.

```
dakota_restart_util print dakota.rst > dakota.txt
```

may be needed to manage the output.

18.2.2 To/From Neutral File Format

A DAKOTA restart file can be converted to a neutral file format using a command like the following:

```
dakota_restart_util to_neutral dakota.rst dakota.neu
```

which results in a report similar to:

```
Writing neutral file dakota.neu
Restart file processing completed: 65 evaluations retrieved.
```

Similarly, a neutral file can be returned to binary format using a command like the following:

```
dakota_restart_util from_neutral dakota.neu dakota.rst
```

which results in a report similar to:

```
Reading neutral file dakota.neu
Writing new restart file dakota.rst
Neutral file processing completed: 65 evaluations retrieved.
```

The contents of the generated neutral file are similar to the following (from the `cyl_head` example problem):

```
Fundamental 2 1.8000000000000000e+00 intake_dia 1.0000000000000000e+00 flatness
0 0 0 0 0 0
NULL 2 4 1 0 1 1 1 1 -2.4355973813000000e+00 -4.7428486676999998e-01
-4.5000000000000001e-01 1.3971143170000000e-01 1
Fundamental 2 1.8001800000000001e+00 intake_dia 1.0000000000000000e+00 flatness
0 0 0 0 0 0
NULL 2 4 1 0 1 1 1 1 -2.4356759411000000e+00 -4.7425991059000000e-01
-4.5000000000000001e-01 1.3971143170000000e-01 2
Fundamental 2 1.7998200000000000e+00 intake_dia 1.0000000000000000e+00 flatness
0 0 0 0 0 0
NULL 2 4 1 0 1 1 1 1 -2.4355188216000001e+00 -4.7430978909999999e-01
-4.5000000000000001e-01 1.3971143170000000e-01 3
... <<omission>> ...
```

This format is not intended for direct viewing (`print` should be used for this purpose). Rather, the neutral file capability has been used in the past for managing portability of restart data across platforms. Recent use of the XDR standard for portable binary formats has eliminated this need, and neutral file conversions may be phased out in future releases.

18.2.3 To Tabular Format

Conversion of a binary restart file to a tabular format enables convenient import of this data into 3rd-party post-processing tools such as Matlab, TECplot, Excel, etc. This facility is nearly identical to the `tabular_graphics_data` option in the DAKOTA input file specification (described in Section 7.3), but with two important differences:

1.

2. No function evaluations are suppressed as they are with `tabular_graphics_data` (i.e., any internal finite difference evaluations are included).

3. The conversion can be performed posthumously, i.e., for DAKOTA runs executed previously.

An example command for converting a restart file to tabular format is:

```
dakota_restart_util to_tabular dakota.rst dakota.m
```

which results in a report similar to:

```
Writing tabular text file dakota.m
Restart file processing completed: 10 evaluations retrieved
     and history of 5 attributes recorded.
```

The contents of the generated tabular file are similar to the following (taken from the `textbook` example problem):

```
% eval_id            x1            x2            f1              f2              f3
        1           0.9           1.1        0.0002            0.26            0.76
        2  0.6433962264  0.6962264151  0.0246865569   0.06584549663    0.1630331079
        3  0.5310576935  0.5388046558 0.09360081618   0.01261994596   0.02478161032
        4   0.612538853  0.6529854907 0.03703861037   0.04871110112    0.1201206246
        5  0.5209215947  0.5259311717  0.1031862798  0.008393722022   0.01614279999
        6  0.5661606434  0.5886684401 0.06405197568   0.02620365411   0.06345021064
        7  0.5083873357   0.510239856  0.1159458957   0.00333775509  0.006151042806
        8  0.5001577143  0.5001800249  0.1248312163 6.772666378e-05  0.000101200204
        9  0.5000000547  0.5000000597  0.1249999428   2.4865003e-08 3.238000351e-08
       10           0.5           0.5         0.125               0               0
```

18.2.4 Concatenation of Multiple Restart Files

In some instances, it is useful to combine restart files into a single master function evaluation database. For example, when constructing a data fit surrogate model, data from previous studies can be pulled in and reused to create a combined data set for the surrogate fit. An example command for concatenating multiple restart files is:

```
dakota_restart_util cat dakota.rst.1 dakota.rst.2 dakota.rst.3 dakota.rst.all
```

which results in a report similar to:

```
Writing new restart file dakota.rst.all
dakota.rst.1 processing completed: 10 evaluations retrieved.
dakota.rst.2 processing completed: 110 evaluations retrieved.
dakota.rst.3 processing completed: 65 evaluations retrieved.
```

The `dakota.rst.all` database now contains 185 evaluations and can be read in for use in a subsequent DAKOTA study using the `-read_restart` option to the `dakota` executable (see Section 18.1).

18.2.5 Removal of Corrupted Data

On occasion, a simulation or computer system failure may cause a corruption of the DAKOTA restart file. For example, a simulation crash may result in failure of a post-processor to retrieve meaningful data. If 0's (or other erroneous data) are returned from the user's `analysis_driver`, then this bad data will get recorded in the restart file. If there is a clear demarcation of where corruption initiated (typical in a process with feedback, such as gradient-based optimization), then use of the `-stop_restart` option for the `dakota` executable can be effective in continuing the study from the point immediately prior to the introduction of bad data. If, however, there are interspersed corruptions throughout the restart database (typical in a process without feedback, such as sampling), then the `remove` and `remove_ids` options of `dakota_restart_util` can be useful.

An example of the command syntax for the `remove` option is:

```
dakota_restart_util remove 2.e-04 dakota.rst dakota.rst.repaired
```

which results in a report similar to:

```
Writing new restart file dakota.rst.repaired
Restart repair completed: 65 evaluations retrieved, 2 removed, 63 saved.
```

where any evaluations in `dakota.rst` having an active response function value that matches `2.e-04` within machine precision are discarded when creating `dakota.rst.repaired`.

An example of the command syntax for the `remove_ids` option is:

```
dakota_restart_util remove_ids 12 15 23 44 57 dakota.rst dakota.rst.repaired
```

which results in a report similar to:

```
Writing new restart file dakota.rst.repaired
Restart repair completed: 65 evaluations retrieved, 5 removed, 60 saved.
```

where evaluation ids `12`, `15`, `23`, `44`, and `57` have been discarded when creating `dakota.rst.repaired`. An important detail of the `remove_ids` option is that evaluations are removed based on the evaluation id that is recorded as part of every restart record, not based on the order of their appearance in the restart file (note: this is the opposite case from that of the `-stop_restart` option described in Section 18.1). This distinction is important when removing restart records for a run that contained either asynchronous or duplicate evaluations, since the restart insertion order and evaluation ids may not correspond in these cases (asynchronous evaluations have ids assigned in the order of job initiation but are recorded in the restart file in the order of job completion, and duplicate evaluations are not recorded which introduces offsets between evaluation id and record number). This can also be important if removing records from a concatenated restart file, since the same evaluation id could appear more than once. In this case, all evaluation records with ids matching the `remove_ids` list will be removed.

Chapter 19

Simulation Code Failure Capturing

DAKOTA provides the capability to manage failures in simulation codes within its system call, fork, and direct application interfaces (see Chapter 5 for application interface descriptions). Failure capturing consists of three operations: failure detection, failure communication, and failure recovery.

19.1 Failure detection

Since the symptoms of a simulation failure are highly code and application dependent, it is the user's responsibility to detect failures within their `analysis_driver` or `output_filter`. One popular example of simulation monitoring is to rely on a simulation's internal detection of errors. In this case, the UNIX `grep` utility can be used within a user's script to detect strings in output files which indicate analysis failure. For example, the following C shell script excerpt

```
grep ERROR analysis.out > /dev/null
if ( $status == 0 )
  echo "FAIL" > results.out
endif
```

will pass the `if` test and communicate simulation failure to DAKOTA if the `grep` command finds the string `ERROR` anywhere in the `analysis.out` file. The `/dev/null` device file is called the "bit bucket" and the `grep` command output is discarded by redirecting it to this destination. The `$status` shell variable contains the exit status of the last command executed [1], which is the exit status of `grep` in this case (0 if successful in finding the error string, nonzero otherwise). For Bourne shells [7], the `$?` shell variable serves the same purpose as `$status` for C shells. In a related approach, if the return code from a simulation can be used directly for failure detection purposes, then `$status` or `$?` could be queried immediately following the simulation execution using an `if` test like that shown above.

If the simulation code is not returning error codes or providing direct error diagnostic information, then failure detection may require monitoring of simulation results for sanity (e.g., is the mesh distorting excessively?) or potentially monitoring for continued process existence to detect a simulation segmentation fault or core dump. While this can get complicated, the flexibility of DAKOTA's interfaces allows for a wide variety of monitoring approaches.

19.2 Failure communication

Once a failure is detected, it must be communicated so that DAKOTA can attempt to recover from the failure. The form of this communication depends on the type of application interface in use.

In the system call and fork application interfaces, a detected simulation failure is communicated to DAKOTA through the results file. Instead of returning the standard results file data, the string "`fail`" should appear at the beginning of the results file. Any data appearing after the fail string will be ignored. Also, DAKOTA's detection of this string is case insensitive, so "`FAIL`", "`Fail`", etc., are equally valid.

In the direct application interface case, a detected simulation failure is communicated to DAKOTA through the return code provided by the user's `analysis_driver`. The prototype for simulations linked within the direct interface is

```
    int analysis_driver(const DakotaVariables& vars,
         const DakotaIntArray& asv, DakotaResponse& response)
```

The `int` returned is the failure code: 0 (false) if no failure occurs and 1 (true) if a failure occurs. Refer to Section 16.2 for additional information on the direct application interface.

19.3 Failure recovery

Once the analysis failure has been communicated, DAKOTA will attempt to recover from the failure using one of the following four mechanisms, as governed by specifications from the interface keyword block in the user's input file (see the DAKOTA Reference Manual [17] for additional information on this specification).

19.3.1 Abort (default)

If the `abort` option is active (the default), then DAKOTA will terminate upon detecting a failure. Note that if the problem causing the failure can be corrected, DAKOTA's restart capability (see Chapter 18) can be used to continue the study.

19.3.2 Retry

If the `retry` option is specified, then DAKOTA will re-invoke the failed simulation up to the specified number of retries. If the simulation continues to fail on each of these retries, DAKOTA will terminate. The retry option is appropriate for those cases in which simulation failures may be resulting from transient computing environment issues, such as shared disk space, software license access, or networking problems.

19.3.3 Recover

If the `recover` option is specified, then DAKOTA will not attempt the failed simulation again. Rather, it will return a "dummy" set of function values as the results of the function evaluation. The dummy function values to be returned are specified by the user. Any gradient or Hessian data requested in the active set vector will be zero. This option is appropriate for those cases in which a failed simulation may indicate a region of the design space to be avoided and the dummy values can be used to return a large objective function or a constraint violation which will discourage an optimizer from further investigating the region.

19.3.4 Continuation

If the `continuation` option is specified, then DAKOTA will attempt to step towards the failing “target” simulation from a nearby “source” simulation through the use of a continuation algorithm. This option is appropriate for those cases in which a failed simulation may be caused by an inadequate initial guess. If the “distance” between the source and target can be divided into smaller steps in which information from one step provides an adequate initial guess for the next step, then the continuation method can step towards the target in increments sufficiently small to allow for convergence of the simulations.

When the failure occurs, the interval between the last successful evaluation (the source point) and the current target point is halved and the evaluation is retried. This halving is repeated until a successful evaluation occurs. The algorithm then marches towards the target point using the last interval as a step size. If a failure occurs while marching forward, the interval will be halved again. Each invocation of the continuation algorithm is allowed a total of ten failures (ten halvings result in up to 1024 evaluations from source to target) prior to aborting the DAKOTA process.

While DAKOTA manages the interval halving and function evaluation invocations, the user is responsible for managing the initial guess for the simulation program. For example, in a GOMA input file [60], the user specifies the files to be used for reading initial guess data and writing solution data. When using the last successful evaluation in the continuation algorithm, the translation of initial guess data can be accomplished by simply copying the solution data file leftover from the last evaluation to the initial guess file for the current evaluation (and in fact this is useful for all evaluations, not just continuation). However, techniques are under development for use of the closest, previously successful, function evaluation (rather than the last successful evaluation) as the source point in the continuation algorithm. This will be especially important for nonlocal methods (e.g., genetic algorithms) in which the last successful evaluation may not necessarily be in the vicinity of the current evaluation. This approach will require the user to save and manipulate previous solutions (likely tagged with evaluation number) so that the results from a particular simulation (specified by DAKOTA after internal identification of the closest point) can be used as the current simulation’s initial guess.

Chapter 20

Additional Examples

20.1 Textbook Example

The optimization problem formulation is stated as

$$\begin{aligned} \texttt{minimize} \quad & f = \sum_{i=1}^{n} (x_i - 1)^4 && (20.1) \\ \texttt{subject to} \quad & g_1 = x_1^2 - \frac{x_2}{2} \le 0 && (20.2) \\ & g_1 = x_2^2 - \frac{x_1}{2} \le 0 \\ & 0.5 \le x_1 \le 5.8 \\ & -2.9 \le x_2 \le 2.9 \end{aligned}$$

where `n` is the number of design variables. The objective function is designed to accommodate an arbitrary number of design variables in order to allow flexible testing of a variety of data sets. Contour plots for the `n=2` case have been shown previously in Figure 2.3 and Figure 2.4.

This example problem may also be used to exercise least squares solution methods by modifying the problem formulation to:

$$\texttt{minimize} \;\; (f)^2 + (g_1)^2 + (g_2)^2 \qquad (20.3)$$

This modification is performed by simply changing the responses specification for the three functions from `num_objective_functions = 1` and `num_nonlinear_inequality_constraints = 2` to `num_least_squares_terms = 3`. Note that the two problem formulations are not equivalent and will have different solutions.

Another way to exercise the least squares methods which would be equivalent to the optimization formulation would be to select the residual functions to be $(x_i - 1)^2$. However, this formulation requires modification to `text_book.C` and will not be presented here. Equation (20.3), on the other hand, can use the existing `text_book.C` without modification. Refer to Section 20.2 for an example of minimizing the same objective function using both optimization and least squares approaches.

20.1.1 Methods

CONMIN, DOT, NPSOL, and OPT++ methods may be used to solve this optimization problem with or without constraints. OPT++ Gauss-Newton and NLSSOL methods may be used to solve the least squares problem.

The `dakota_textbook.in` file provided in the `Dakota/test` directory selects a `dot_mmfd` optimizer to perform constrained minimization using the `text_book` simulator.

A multilevel hybrid can also be demonstrated on the `text_book` problem. The `dakota_multilevel.in` file provided in the `Dakota/test` directory starts with a `coliny_ea` solution which feeds its best point into a `coliny_pattern_search` optimization which feeds its best point into `optpp_newton`. While this approach is overkill for such a simple problem, it is useful for demonstrating the coordination between multiple methods in the multilevel strategy.

In addition, `dakota_textbook_3pc.in` demonstrates the use of a 3-piece interface to perform the parameter to response mapping and `dakota_textbook_lhs.in` demonstrates the use of latin hypercube Monte Carlo sampling for assessing probability of failure as measured by specified response thresholds.

20.1.2 Optimization Results

The solution for the unconstrained optimization problem for two design variables is:

x_1 = 1.0
x_2 = 1.0

with

f^* = 0.0

The solution for the optimization problem constrained by g_1 is:

x_1 = 0.763
x_2 = 1.16

with

f^* = 0.00388
g_1^* = 0.0 (active)

The solution for the optimization problem constrained by g_1 and g_2 is:

x_1 = 0.500
x_2 = 0.500

with

f^* = 0.125
g_1^* = 0.0 (active)
g_2^* = 0.0 (active)

Note that as constraints are added, the design freedom is restricted and a penalty in the objective function is observed. Of course, no penalty would be observed if the additional constraints were not active at the solution.

20.1.3 Least Squares Results

The solution for the least squares problem is:

```
x1  = 0.566
x2  = 0.566
```

with the residual functions equal to

```
f*  = 0.0713
g1*  = 0.0371
g2*  = 0.0371
```

and a minimal sum of the squares of `0.00783`.

This study requires selection of `num_least_squares_terms = 3` in the responses specification and selection of either `optpp_g_newton` or `nlssol_sqp` in the method specification.

20.2 Rosenbrock Example

The Rosenbrock function [32] is a well known benchmark problem for optimization algorithms. Its formulation can be stated as

$$\texttt{minimize} \;\; f = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{20.4}$$

Surface and contour plots for this function have been shown previously in Figure 2.1 and Figure 2.2. This example problem may also be used to exercise least squares solution methods by recasting the problem formulation into:

$$\texttt{minimize} \;\; f = (f_1)^2 + (f_2)^2 \tag{20.5}$$

where

$$f_1 = 10(x_2 - x_1^2) \tag{20.6}$$

and

$$f_2 = 1 - x_1 \tag{20.7}$$

are residual terms. In this case (unlike the least squares modification in Section 20.1), the two problem formulations are equivalent and have identical solutions.

20.2.1 Methods

In the /Dakota/test directory, the rosenbrock executable (compiled from rosenbrock.C) returns an objective function as computed from Equation (20.4) for use with optimization methods. The rosenbrock_ls executable (compiled from rosenbrock_ls.C) returns two least squares terms as computed from Equation (20.6) and Equation (20.7) for use with least squares methods. Both executables return analytic gradients of the function set (gradient of the objective function in rosenbrock, gradients of the least squares residuals in rosenbrock_ls) with respect to the design variables. The dakota_rosenbrock.in input file can be used to solve both problems by toggling settings in the interface, responses, and method specifications. To run the optimization solution, select 'rosenbrock' as the analysis_driver in the interface specification, select num_objective_functions = 1 in the responses specification, and select an optimizer (e.g., optpp_q_newton) in the method specification, e.g.:

```
interface,                                              \
       application system                               \
         analysis_driver = 'rosenbrock'

variables,                                              \
       continuous_design = 2                            \
        cdv_initial_point   -1.2      1.0               \
        cdv_lower_bounds    -2.0     -2.0               \
        cdv_upper_bounds     2.0      2.0               \
        cdv_descriptor       'x1'     'x2'

responses,                                              \
       num_objective_functions = 1                      \
       analytic_gradients                               \
       no_hessians

method,                                                 \
       optpp_q_newton                                   \
         convergence_tolerance = 1e-10
```

To run the least squares solution, select 'rosenbrock_ls' as the analysis_driver in the interface specification, select num_least_squares_terms = 2 in the responses specification, and select a least squares iterator (i.e., optpp_g_newton or nlssol_sqp) in the method specification, e.g.:

```
interface,                                              \
        application system                              \
          analysis_driver = 'rosenbrock_ls'

variables,                                              \
        continuous_design = 2                           \
          cdv_initial_point   -1.2      1.0             \
          cdv_lower_bounds    -2.0     -2.0             \
          cdv_upper_bounds     2.0      2.0             \
          cdv_descriptor       'x1'     'x2'

responses,                                              \
        num_least_squares_terms = 2                     \
        analytic_gradients                              \
        no_hessians

method,                                                 \
        optpp_g_newton,                                 \
          convergence_tolerance = 1e-10
```

20.2.2 Results

The optimal solution, solved either as a least squares problem or an optimization problem, is:

$$x_1 = 1.0$$
$$x_2 = 1.0$$

with

$$f^* = 0.0$$

In comparing the two approaches, one would expect the Gauss-Newton approach to be more efficient since it exploits the special-structure of a least squares objective function. From a good initial guess, this expected behavior is observed. Starting from `cdv_initial_point = 0.8, 0.7`, the `optpp_g_newton` method converges in only 3 function and gradient evaluations while the `optpp_q_newton` method requires 14 function and gradient evaluations to achieve similar accuracy. Starting from a poorer initial guess (e.g., `cdv_initial_point = -1.2, 1.0` as specified in `Dakota/test/dakota_rosenbrock.in`), the trend is less obvious since both methods spend several evaluations finding the vicinity of the minimum (total function and gradient evaluations = 24 for `optpp_q_newton` and 29 for `optpp_g_newton`). However, once the vicinity is located, convergence is much more rapid with the Gauss-Newton approach (11 orders of magnitude reduction in the objective function in 1 function and gradient evaluation) than with the quasi-Newton approach (12 orders of magnitude reduction in the objective function in 10 function and gradient evaluations).

Shown below is the complete DAKOTA output for the `optpp_g_newton` method starting from `cdv_initial_point = 0.8, 0.7`:

```
Running MPI executable in serial mode.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
methodName = optpp_g_newton
gradientType = analytic
hessianType = none

>>>>> Running Single Method Strategy.

>>>>> Running optpp_g_newton iterator.

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      8.0000000000e-01 x1
                      7.0000000000e-01 x2

(rosenbrock_ls /tmp/fileYHvTvE /tmp/file6f8NR8)

Active response data for function evaluation 1:
Active set vector = { 3 3 }
                      6.0000000000e-01 least_sq_term1
                      2.0000000000e-01 least_sq_term2
 [ -1.6000000000e+01  1.0000000000e+01  ] least_sq_term1 gradient
 [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient
```

```
    nlf2_evaluator_gn results: objective fn. =
    4.0000000000e-01
    nlf2_evaluator_gn results: objective fn. gradient =
    [ -1.9600000000e+01  1.2000000000e+01 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  5.1400000000e+02 -3.2000000000e+02
   -3.2000000000e+02  2.0000000000e+02 ]]


------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                      9.9999528206e-01 x1
                      9.5999243139e-01 x2

(rosenbrock_ls /tmp/fileaS7ICC /tmp/fileymcJm6)

Active response data for function evaluation 2:
Active set vector = { 3 3 }
                     -3.9998132752e-01 least_sq_term1
                      4.7179400000e-06 least_sq_term2
 [ -1.9999905641e+01  1.0000000000e+01  ] least_sq_term1 gradient
 [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient


    nlf2_evaluator_gn results: objective fn. =
    1.5998506239e-01
    nlf2_evaluator_gn results: objective fn. gradient =
 [  1.5999168181e+01 -7.9996265504e+00 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  8.0199245130e+02 -3.9999811282e+02
 -3.9999811282e+02  2.0000000000e+02 ]]


------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                      9.9999904378e-01 x1
                      9.9999808275e-01 x2

(rosenbrock_ls /tmp/fileSAHG0B /tmp/fileK8lAE7)

Active response data for function evaluation 3:
Active set vector = { 3 3 }
                     -4.8109144216e-08 least_sq_term1
                      9.5621999996e-07 least_sq_term2
 [ -1.9999980876e+01  1.0000000000e+01  ] least_sq_term1 gradient
 [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient


    nlf2_evaluator_gn results: objective fn. =
    9.1667117808e-13
    nlf2_evaluator_gn results: objective fn. gradient =
 [  1.1923928641e-08 -9.6218288432e-07 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  8.0199847008e+02 -3.9999961752e+02
```

```
 -3.9999961752e+02  2.0000000000e+02 ]]


<<<<< Iterator optpp_g_newton completed.
<<<<< Function evaluation summary: 3 total (3 new, 0 duplicate)
<<<<< Best parameters        =
                     9.9999904378e-01 x1
                     9.9999808275e-01 x2
<<<<< Best residual terms     =
                    -4.8109144216e-08
                     9.5621999996e-07
<<<<< Best data captured at function evaluation 3
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =      0.01 [parent =  0.009765, child =  0.000235]
  Total wall clock =     0.026
```

20.3 Cylinder Head Example

The cylinder head example problem arose as a simple demonstration problem for the Technologies Enabling Agile Manufacturing (TEAM) project. Its formulation is stated as

$$\begin{aligned}
\texttt{minimize} \quad & f = -1\left(\frac{\texttt{horsepower}}{250} + \frac{\texttt{warranty}}{100000}\right) && (20.8)\\
\texttt{subject to} \quad & \sigma_{max} \le 0.5\sigma_{yield} && (20.9)\\
& \texttt{warranty} \ge 100000\\
& \texttt{time}_{\texttt{cycle}} \le 60\\
& 1.5 \le \texttt{d}_{\texttt{intake}} \le 2.164\\
& 0.0 \le \texttt{flatness} \le 4.0
\end{aligned}$$

This formulation seeks to simultaneously maximize normalized engine horsepower and engine warranty over variables of valve intake diameter ($\texttt{d}_{\texttt{intake}}$) in inches and overall head flatness (flatness) in thousandths of an inch subject to inequality constraints that the maximum stress cannot exceed half of yield, that warranty must be at least 100000 miles, and that manufacturing cycle time must be less than 60 seconds. Since the constraints involve different scales, they should be nondimensionalized. In addition, they can be converted to the standard 1-sided form $g(\mathbf{x}) \le 0$ as follows:

$$\begin{aligned}
g_1 &= \frac{2\sigma_{\texttt{max}}}{\sigma_{\texttt{yield}}} - 1 \le 0\\
g_2 &= 1 - \frac{\texttt{warranty}}{100000} \le 0 \qquad (20.10)\\
g_3 &= \frac{\texttt{time}_{\texttt{cycle}}}{60} - 1 \le 0
\end{aligned}$$

The objective function and constraints are related analytically to the design variables according to the following simple expressions:

$$\begin{aligned}
\texttt{warranty} &= 100000 + 15000(4 - \texttt{flatness})\\
\texttt{time}_{\texttt{cycle}} &= 45 + 4.5(4 - \texttt{flatness})^{1.5}
\end{aligned}$$

$$\begin{aligned}
\texttt{horsepower} &= 250 + 200\left(\frac{\texttt{d}_{\texttt{intake}}}{1.833} - 1\right) \\
\sigma_{\texttt{max}} &= 750 + \frac{1}{(\texttt{t}_{\texttt{wall}})^{2.5}} \\
\texttt{t}_{\texttt{wall}} &= \texttt{offset}_{\texttt{intake}} - \texttt{offset}_{\texttt{exhaust}} - \frac{(\texttt{d}_{\texttt{intake}} - \texttt{d}_{\texttt{exhaust}})}{2}
\end{aligned} \tag{20.11}$$

where the constants in Equation (20.10) and Equation (20.11) assume the following values: $\sigma_{\texttt{yield}} = 3000$, $\texttt{offset}_{\texttt{intake}} = 3.25$, $\texttt{offset}_{\texttt{exhaust}} = 1.34$, and $\texttt{d}_{\texttt{exhaust}} = 1.556$.

20.3.1 Methods

In the `Dakota/test` directory, the `dakota_cyl_head.in` input file is used to execute the cylinder head example. This input file manages a variety of tests, of which one is shown below:

```
interface,                                                    \
        application fork,                                     \
          asynchronous                                        \
          analysis_driver=  'cyl_head'

variables,                                                    \
        continuous_design = 2                                 \
          cdv_initial_point    1.8    1.0                     \
          cdv_upper_bounds     2.164  4.0                     \
          cdv_lower_bounds     1.5    0.0                     \
          cdv_descriptor 'intake_dia' 'flatness'

responses,                                                    \
        num_objective_functions = 1                           \
        num_nonlinear_inequality_constraints = 3              \
        numerical_gradients                                   \
          method_source dakota                                \
          interval_type central                               \
          fd_step_size = 1.e-4                                \
        no_hessians

method,                                                       \
        npsol_sqp                                             \
          convergence_tolerance = 1.e-8                       \
          output verbose
```

The interface keyword specifies use of the `cyl_head` executable (compiled from `/Dakota/test/cyl_head.C`) as the simulator. The variables and responses keywords specify the data sets to be used in the iteration by providing the initial point, descriptors, and upper and lower bounds for two continuous design variables and by specifying the use of one objective function, three inequality constraints, and numerical gradients in the problem. The method keyword specifies the use of the `npsol_sqp` method to solve this constrained optimization problem. No strategy keyword is specified, so the default `single_method` strategy is used.

20.3.2 Optimization Results

The solution for the constrained optimization problem is:

```
intake_dia = 2.122
flatness   = 1.769
```

with

```
f* = -2.461
g1* = 0.0 (active)
g2* = -0.3347 (inactive)
g3* = 0.0 (active)
```

which corresponds to the following optimal response quantities:

```
warranty = 133472
cycle_time = 60
horse_power = 281.579
max_stress = 1500
```

The final report from the DAKOTA output is as follows:

```
<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 65 total (65 new, 0 duplicate)
<<<<< Best parameters        =
                      2.1224188321e+00 intake_dia
                      1.7685568330e+00 flatness
<<<<< Best objective function =
                     -2.4610312954e+00
<<<<< Best constraint values  =
                     -5.3569116666e-10
                     -3.3471647505e-01
                      9.9882176098e-12
<<<<< Best data captured at function evaluation 61
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
 Total CPU       =      0.06 [parent =  0.068359, child = -0.008359]
 Total wall clock =     0.212
```

20.4 Container Example

For this example, suppose that a high-volume manufacturer of light weight steel containers wants to minimize the amount of raw sheet material that must be used to manufacture a 1.1 quart cylindrical-shaped can, including waste material. Material for the container walls and end caps is stamped from stock sheet material of constant thickness. The seal between the end caps and container wall is manufactured by a press forming operation on the end caps. The end caps can then be attached to the container wall forming a seal through a crimping operation.

For preliminary design purposes, the extra material that would normally go into the container end cap seals is approximated by increasing the cut dimensions of the end cap diameters by 12% and the height of the container wall by 5%, and waste associated with stamping the end caps in a specialized pattern from sheet stock is estimated as 15% of the cap area. The equation for the area of the container materials including waste is

$$A = 2\times\left(\begin{array}{c}\texttt{end cap}\\\texttt{waste}\\\texttt{material}\\\texttt{factor}\end{array}\right)\times\left(\begin{array}{c}\texttt{end cap}\\\texttt{seal}\\\texttt{material}\\\texttt{factor}\end{array}\right)\times\left(\begin{array}{c}\texttt{nominal}\\\texttt{end cap}\\\texttt{area}\end{array}\right)+\left(\begin{array}{c}\texttt{container}\\\texttt{wall seal}\\\texttt{material}\\\texttt{factor}\end{array}\right)\times\left(\begin{array}{c}\texttt{nominal}\\\texttt{container}\\\texttt{wall area}\end{array}\right)$$

or

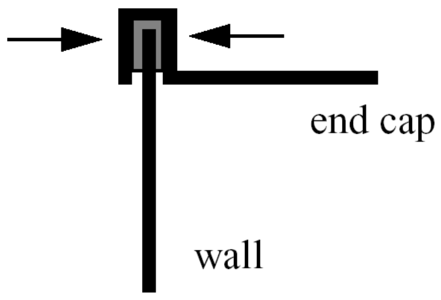


Figure 20.1: Container wall-to-end-cap seal

$$A = 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)\pi DH \tag{20.12}$$

where D and H are the diameter and height of the finished product in units of inches, respectively. The volume of the finished product is given by

$$V = \pi\frac{D^2H}{4} = (1.1\texttt{qt})(57.75\texttt{in}^3/\texttt{qt}) \tag{20.13}$$

The equation for area is the objective function for this problem; it is to be minimized. The equation for volume is an equality constraint; it must be satisfied at the conclusion of the optimization problem. Any combination of D and H that satisfies the volume constraint is a **feasible** solution (although not necessarily the optimal solution) to the area minimization problem, and any combination that does not satisfy the volume constraint is an **infeasible** solution. The area that is a minimum subject to the volume constraint is the **optimal** area, and the corresponding values for the parameters D and H are the optimal parameter values.

It is important that the equations supplied to a numerical optimization code be limited to generating only physically realizable values, since an optimizer will not have the capability to differentiate between meaningful and nonphysical parameter values. It is often up to the engineer to supply these limits, usually in the form of parameter bound constraints. For example, by observing the equations for the area objective function and the volume constraint, it can be seen that by allowing the diameter, D, to become negative, it is algebraically possible to generate relatively small values for the area that also satisfy the volume constraint. Negative values for D are of course physically meaningless. Therefore, to ensure that the numerically-solved optimization problem remains meaningful, a bound constraint of $D \leq 0$ must be included in the optimization problem statement. A positive value for H is implied since the volume constraint could never be satisfied if H were negative. However, a bound constraint of $H \leq 0$ can be added to the optimization problem if desired. The optimization problem can then be stated in a standardized form as

$$\begin{aligned} \texttt{minimize:} \quad & 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)^2\pi DH \\ \texttt{subject to:} \quad & \pi\frac{D^2H}{4} = (1.1\texttt{qt})(57.75\texttt{in}^3/\texttt{qt}) \\ & D \leq 0, H \leq 0 \end{aligned} \tag{20.14}$$

A graphical view of the container optimization problem appears in Figure 20.2. The 3-D surface defines the area, A, as a function of diameter and height. The curved line that extends across the surface defines the areas that satisfy the volume equality constraint, V. Graphically, the container optimization problem can be viewed as one of finding the point along the constraint line with the smallest 3-D surface height in Figure 20.2. This point corresponds to the optimal values for diameter and height of the final product.

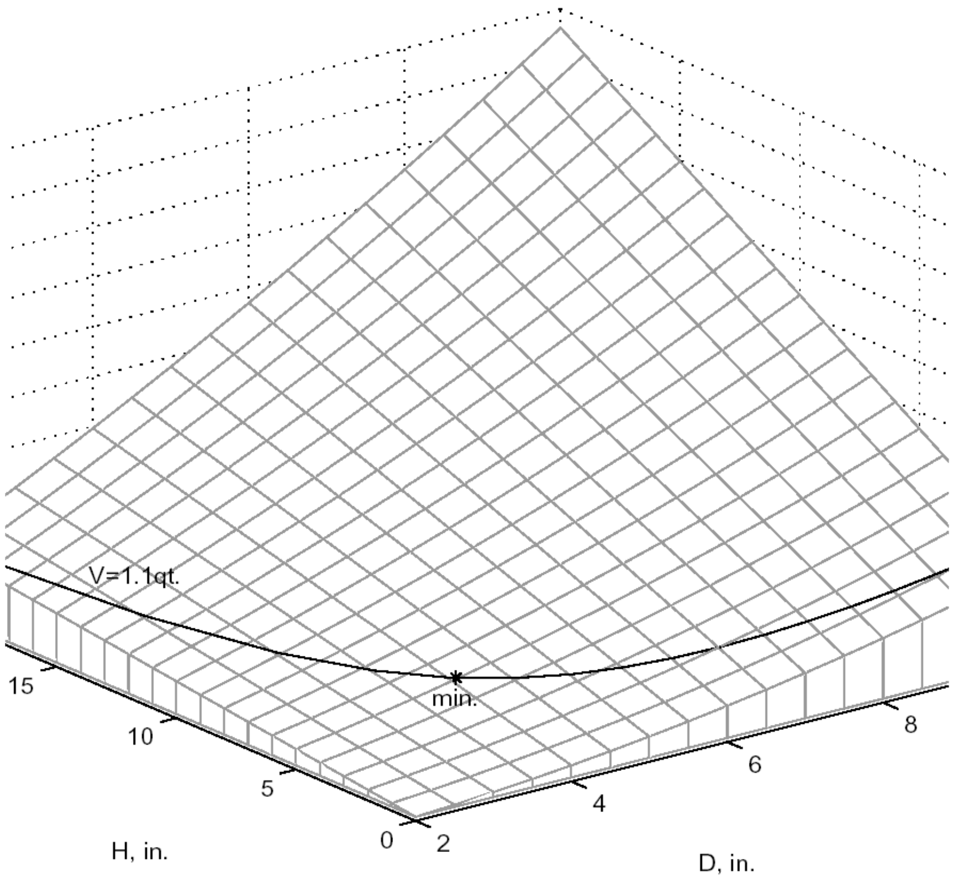


Figure 20.2: A graphical representation of the container optimization problem.

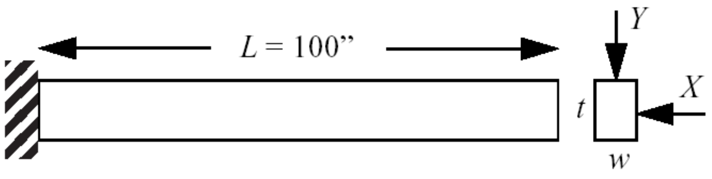


Figure 20.3: Cantilever beam test problem.

The input file for this test problem is named `dakota_container.in` in the directory `/Dakota/test`. The solution to this example problem is $(H, D) = (4.99, 4.03)$, with a minimum area of 98.43 $\texttt{in}^2$.

The final report from the DAKOTA output is as follows:

```
<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 40 total (40 new, 0 duplicate)
<<<<< Best parameters  =
                      4.9873894231e+00 H
                      4.0270846274e+00 D
<<<<< Best objective function =
                      9.8432498115e+01
<<<<< Best constraint values   =
                     -1.2072307876e-09
<<<<< Best data captured at function evaluation 36
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =      0.05 [parent = 0.05, child =6.93889e-18]
  Total wall clock =      0.311
```

20.5 Cantilever Example

This test problem is adapted from the reliability-based design optimization literature [63], [70] and involves a simple uniform cantilever beam as shown in Figure 20.3.

The design problem is to minimize the weight (or, equivalently, the cross-sectional area) of the beam subject to a displacement constraint and a stress constraint. Random variables in the problem include the yield stress R of the beam material, the Young's modulus E of the material, and the horizontal and vertical loads, X and Y, which are modeled with normal distributions using $N(40000, 2000)$, $N(2.9E7, 1.45E6)$, $N(500, 100)$, and $N(1000, 100)$, respectively. Problem constants include $L = 100\texttt{in}$ and $D_0 = 2.2535\texttt{in}$. The constraints have the following analytic form:

$$\begin{aligned} \texttt{stress} &= \frac{600}{wt^2}Y + \frac{600}{w^2t}X \leq R && (20.15) \\ \texttt{displacement} &= \frac{4L^3}{Ewt}\sqrt{\left(\frac{Y}{t^2}\right)^2 + \left(\frac{X}{w^2}\right)^2} \leq D_0 \end{aligned}$$

or when scaled:

$$\begin{aligned} g_S &= \frac{\texttt{stress}}{R} - 1 \leq 0 && (20.16) \\ g_D &= \frac{\texttt{displacement}}{D_0} - 1 \leq 0 \end{aligned}$$

(20.17)

20.5.1 Deterministic Optimization Results

If the random variables E, R, X, and Y are fixed at their means, the resulting deterministic design problem can be formulated as

$$\begin{aligned} \texttt{minimize} \quad & f = wt && (20.18) \\ \texttt{subject to} \quad & g_S \leq 0 && (20.19) \\ & g_D \leq 0 \\ & 1.0 \leq w \leq 4.0 \\ & 1.0 \leq t \leq 4.0 \end{aligned}$$

and can be solved using the `Dakota/test/dakota_cantilever.in` file. This input file manages a variety of tests, of which one follows:

```
method,                                                          \
        npsol_sqp                                                \
          convergence_tolerance = 1.e-8                          \
          output verbose

variables,                                                       \
        continuous_design = 2                                    \
          cdv_initial_point    4.0          4.0                  \
          cdv_upper_bounds    10.0         10.0                  \
          cdv_lower_bounds     1.0          1.0                  \
          cdv_descriptor      'beam_width' 'beam_thickness'      \
        continuous_state = 4                                     \
          csv_initial_state   40000.  29.E+6  500.  1000.        \
          csv_descriptor        'R'     'E'    'X'    'Y'

interface,                                                       \
        application system,                                      \
          asynchronous evaluation_concurrency = 2                \
          analysis_driver = 'cantilever'

responses,                                                       \
        num_objective_functions = 1                              \
        num_nonlinear_inequality_constraints = 2                 \
        numerical_gradients                                      \
          method_source dakota                                   \
          interval_type forward                                  \
          fd_step_size = 1.e-4                                   \
        no_hessians
```

The deterministic solution is $(w, t) = (2.35, 3.33)$ with an objective function of 7.82. The final report from the DAKOTA output is as follows:

```
<<<<< Iterator npsol_sqp completed.
<<<<< Function evaluation summary: 33 total (33 new, 0 duplicate)
<<<<< Best parameters         =
                      2.3520341345e+00 beam_width
                      3.3262783972e+00 beam_thickness
                      4.0000000000e+04 R
                      2.9000000000e+07 E
                      5.0000000000e+02 X
                      1.0000000000e+03 Y
<<<<< Best objective function =
                      7.8235203311e+00
<<<<< Best constraint values  =
                     -1.6008999688e-02
                      1.9308333361e-11
<<<<< Best data captured at function evaluation 31
<<<<< Single Method Strategy completed.
DAKOTA execution time in seconds:
  Total CPU        =      0.04 [parent =  0.050781, child = -0.010781]
  Total wall clock =     0.299
```

20.5.2 Stochastic Optimization Results

If the normal distributions for the random variables E, R, X, and Y are included, a stochastic design problem can be formulated as

$$\begin{aligned} \texttt{minimize} \quad & f = wt && (20.20) \\ \texttt{subject to} \quad & \mu_D + 3\sigma_D \leq 0 && (20.21) \\ & \mu_S + 3\sigma_S \leq 0 \\ & 1.0 \leq w \leq 4.0 \\ & 1.0 \leq t \leq 4.0 \end{aligned}$$

where a 3-sigma reliability level (probability of failure = 0.00135 if responses are normally-distributed) is being sought on the scaled constraints. Optimization under uncertainty solutions to the stochastic problem are described in [19].

20.6 Multiobjective Examples

There are three examples in the test directory that are taken from a multiobjective evolutionary algorithm (MOEA) test suite described by Van Veldhuizen et. al. in [75]. These three problems are good examples to illustrate different forms the Pareto set may take. For each problem, we list the DAKOTA input file, and show a graph of the Pareto front. These problems are all solved with the `moga` method. In Van Veldhuizen's notation, the set of all Pareto optimal design configurations (design variable values only) is denoted $\mathtt{P}^*$ or $\mathtt{P}_{\mathtt{true}}$ and is defined as:

$$P^* := \{x \in \Omega \,|\, \neg\exists\ x' \in \Omega \quad \bar{f}(x') \preceq \bar{f}(x)\}$$

The Pareto front, which is the set of objective function values associated with the Pareto optimal design configurations, is denoted $\mathtt{PF}^*$ or $\mathtt{PF}_{\mathtt{true}}$ and is defined as:

$$PF^* := \{\bar{u} = \bar{f} = (f_1(x), \ldots, f_k(x)) \,|\, x \in P^*\}$$

The values calculated for the Pareto set and the Pareto front using the moga method are close to but not always exactly the true values, depending on the number of generations the moga is run, the various settings governing the GA, and the complexity of the Pareto set.

20.6.1 Multiobjective Test Problem 1

The first test problem is a case where P_{true} is connected and PF_{true} is concave. The problem is to simultaneously optimize f_1 and f_2 given three input variables, x_1, x_2, and x_3, where the inputs are bounded by $-4 \leq x_i \leq 4$:

$$f_1(x) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i - \frac{1}{\sqrt{3}}\right)^2\right)$$

$$f_2(x) = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i + \frac{1}{\sqrt{3}}\right)^2\right)$$

The input file for this example is the file `/Dakota/test/dakota_mogatest1.in`, see Figure 20.4. The interface keyword specifies the use of the `mogatest1` executable (compiled from `Dakota/test/mogatest1.C`) as the simulator. The Pareto front is shown in Figure 20.5.

20.6.2 Multiobjective Test Problem 2

The second test problem is a case where both $\mathtt{P}_{\mathtt{true}}$ and $\mathtt{PF}_{\mathtt{true}}$ is disconnected. $\mathtt{PF}_{\mathtt{true}}$ has four separate Pareto curves. The problem is to simultaneously optimize f_1 and f_2 given two input variables, x_1 and x_2, where the inputs are bounded by $0 \leq x_i \leq 1$, and:

$$f_1(x) = x_1$$

$$f_2(x) = (1 + 10x_2) \times \left[1 - \left(\frac{x_1}{1 + 10x_2}\right)^2 - \frac{x_1}{1 + 10x_2}\sin(8\pi x_1)\right]$$

The input file for this example is the file `/Dakota/test/dakota_mogatest2.in`, see Figure 20.6. The interface keyword specifies the use of the `mogatest2` executable (compiled from `Dakota/test/mogatest2.C`) as the simulator. The Pareto front is shown in Figure 20.7. Note the discontinous nature of the Pareto front in this example.

20.6.3 Multiobjective Test Problem 3

The third test problem is a case where $\mathtt{P}_{\mathtt{true}}$ is disconnected but $\mathtt{PF}_{\mathtt{true}}$ is connected. It is called the Srinivas problem in the literature (cite). This problem also has two nonlinear constraints. The problem is to simultaneously optimize f_1 and f_2 given two input variables, x_1 and x_2, where the inputs are bounded by $-20 \leq x_i \leq 20$, and:

$$f_1(x) = (x_1 - 2)^2 + (x_2 - 1)^2 + 2$$

$$f_2(x) = 9x_1 - (x_2 - 1)^2$$

The constraints are:

$$0 \leq x_1^2 + x_2^2 - 225$$

```
strategy,                                                       \
        single_method                                           \

method,                                                         \
        moga                                                    \
        output silent                                           \
        seed = 10983                                            \
        max_function_evaluations = 2500                         \
        initialization_type                                     \
                unique_random                                   \
        crossover_type                                          \
                multi_point_parameterized_binary = 3            \
                        crossover_rate = 0.8                    \
        mutation_type                                           \
                replace_uniform                                 \
                        mutation_rate = 0.1                     \
        fitness_type                                            \
                domination_count                                \
        replacement_type                                        \
                below_limit = 6                                 \
                        shrinkage_percentage = 0.9              \
        convergence_type                                        \
                metric_tracker                                  \
                        percent_change = 0.05                   \
                        num_generations = 10

variables,                                              \
        continuous_design = 3                           \
          cdv_initial_point     0        0       0      \
          cdv_upper_bounds      4        4       4      \
          cdv_lower_bounds     -4       -4      -4      \
          cdv_descriptor       'x1'     'x2'    'x3'

interface,                                              \
        system                                          \
          analysis_driver = 'mogatest1'

responses,                                              \
        num_objective_functions = 2                     \
        no_gradients                                    \
        no_hessians
```

Figure 20.4: A DAKOTA input file that specifies the use of a multiple objective genetic algorithm (MOGA) on mogatest1

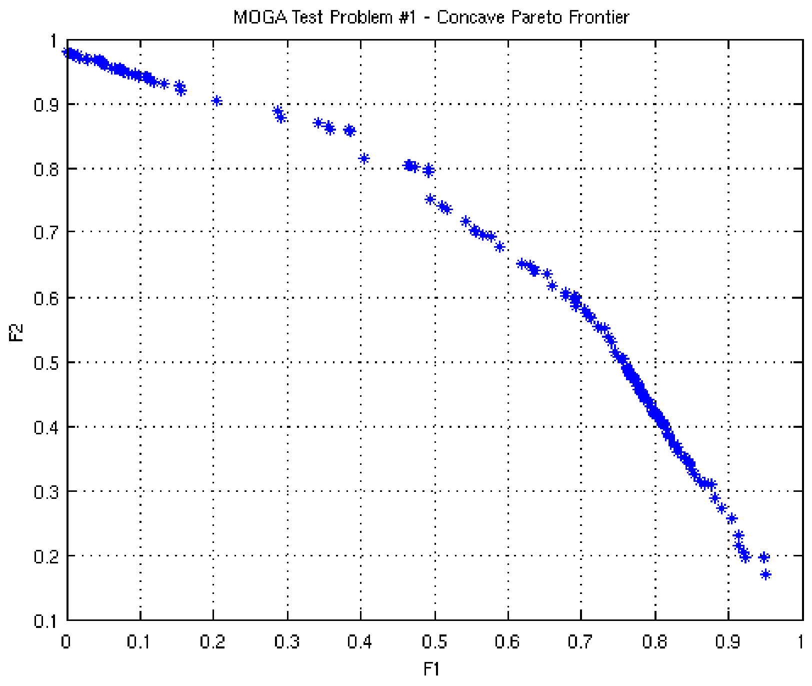


Figure 20.5: Pareto Front showing Tradeoffs between Function F1 and Function F2 for mogatest1

$$0 \leq x_1 - 3x_2 + 10$$

The input file for this example is the file `/Dakota/test/dakota_mogatest3.in`, see Figure 20.8. The interface keyword specifies the use of the `mogatest3` executable (compiled from`Dakota/test/mogatest3.C`) as the simulator. The Pareto set is shown in Figure 20.9. Note the discontinous nature of the Pareto set (in the design space) in this example. The Pareto front is shown in Figure 20.10. Again, note the unusual nature of this Pareto example (these figures agree reasonably well with the Srinivas problem results shown in the literature).

```
strategy,                                                       \
        single                                                  \

method,                                                         \
        moga                                                    \
        output silent                                           \
        seed = 10983                                            \
        max_function_evaluations = 3000                         \
        initialization_type                                     \
                unique_random                                   \
        crossover_type                                          \
                multi_point_parameterized_binary = 3            \
                        crossover_rate = 0.8                    \
        mutation_type                                           \
                replace_uniform                                 \
                        mutation_rate = 0.1                     \
        fitness_type                                            \
                domination_count                                \
        replacement_type                                        \
                below_limit = 6                                 \
                shrinkage_percentage = 0.9                      \
        convergence_type                                        \
                metric_tracker                                  \
                        percent_change = 0.05                   \
                        num_generations = 10                    \

variables,                                              \
        continuous_design = 2                           \
          cdv_initial_point    0.5      0.5             \
          cdv_upper_bounds     1        1               \
          cdv_lower_bounds     0        0               \
          cdv_descriptor       'x1'      'x2'

interface,                                              \
        system                                          \
          analysis_driver = 'mogatest2'

responses,                                              \
        num_objective_functions = 2                     \
        no_gradients                                    \
        no_hessians
```

Figure 20.6: A DAKOTA input file that specifies the use of a multiple objective genetic algorithm (MOGA)

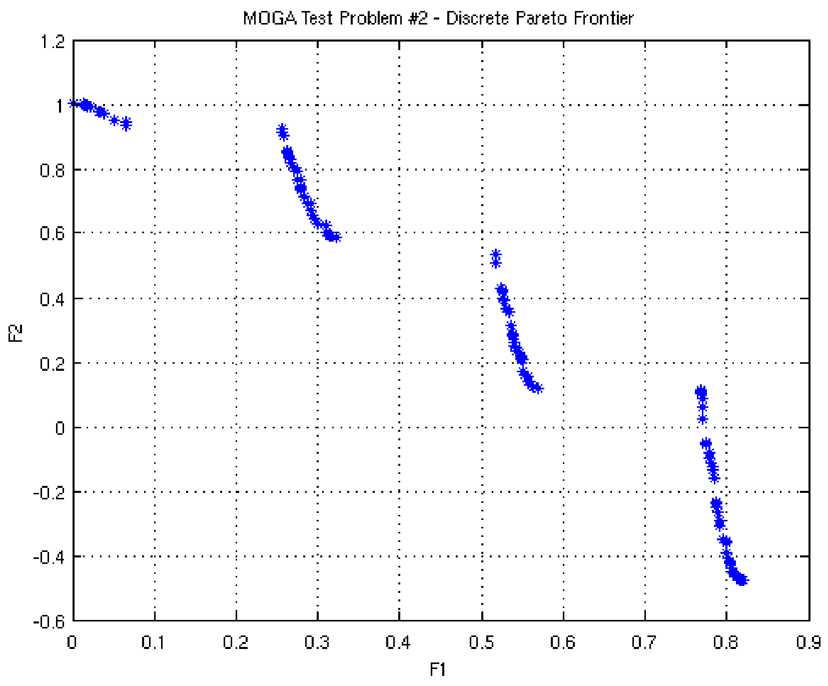


Figure 20.7: Pareto Front showing Tradeoffs between Function F1 and Function F2 for mogatest1

```
strategy,                                                         \
        single                                                    \

method,                                                           \
        moga                                                      \
        output silent                                             \
        seed = 10983                                              \
        max_function_evaluations = 2000                           \
        initialization_type                                       \
                unique_random                                     \
        crossover_type                                            \
                multi_point_parameterized_binary = 3              \
                        crossover_rate = 0.8                      \
        mutation_type                                             \
                replace_uniform                                   \
                        mutation_rate = 0.1                       \
        fitness_type                                              \
                domination_count                                  \
        replacement_type                                          \
                below_limit = 6                                   \
                shrinkage_percentage = 0.9                        \
        convergence_type                                          \
                metric_tracker                                    \
                        percent_change = 0.05                     \
                        num_generations = 10

variables,                                                \
        continuous_design = 2                             \
          cdv_initial_point      0        0               \
          cdv_upper_bounds      20       20               \
          cdv_lower_bounds     -20      -20               \
          cdv_descriptor       'x1'      'x2'

interface,                                                \
        system                                            \
          analysis_driver = 'mogatest3'

responses,                                                \
        num_objective_functions = 2                       \
        num_nonlinear_inequality_constraints = 2          \
        nonlinear_inequality_upper_bounds = 0.0 0.0       \
        no_gradients                                      \
        no_hessians
```

Figure 20.8: A DAKOTA input file that specifies the use of a multiple objective genetic algorithm (MOGA)

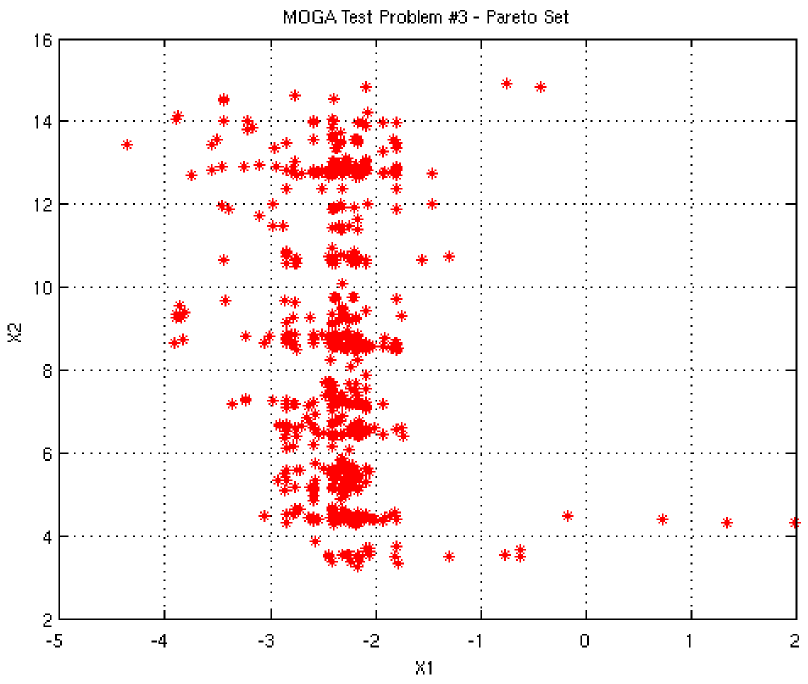


Figure 20.9: Pareto Set of Design Variables corresponding to the Pareto front for mogatest3

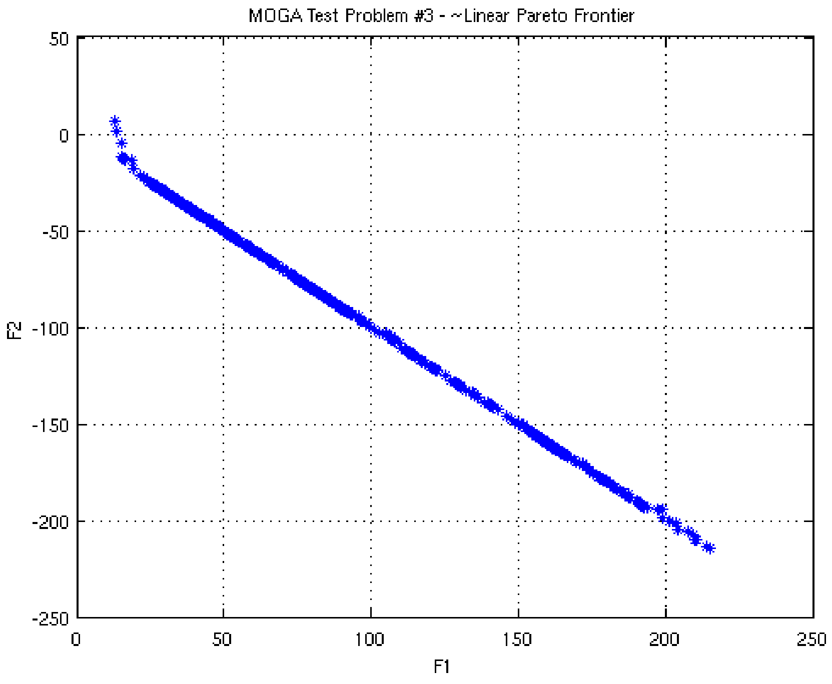


Figure 20.10: Pareto Front showing Tradeoffs between Function F1 and Function F2 for mogatest3

Bibliography

[1] Alexandrov, M., Dennis, J. E., Jr., Lewis, R. M., and Torczon, V.: "A Trust-region Framework for Managing the Use of Approximation Models in Optimization," Structural Optimization, Vol. 15, 1998, pp. 16-23. 23, 74, 75, 157, 183, 205

[2] Anderson, G., and Anderson, P.: *The UNIX C Shell Field Guide*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[3] Arora, J. S.: Introduction to Optimum Design, McGraw-Hill, New York, NY, 1989. 15

[4] Bartlett, R.: Object-Oriented Approaches to Large-Scale NonLinear Programming For Process Systems Engineering, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, 2001. 137

[5] Bartlett, R.A. and Biegler, L.T.: "rSQP++: An Object-Oriented Framework for Successive Quadratic Programming," abstract for First Sandia Workshop on Large-scale PDE-constrained Optimization, Santa Fe, NM, April 4, 2001, to appear in Springer-Verlag Lecture Notes in Computational Science and Engineering. 166

[6] Biros G.: Lagrange-Newton-Krylov-Schur methods for PDE-constrained optimization, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, 2000. 137

[7] Blinn, B.: Portable Shell Programming: An Extensive Collection of Bourne Shell Examples, Prentice Hall PTR, New Jersey, 1996. 183, 205

[8] Byrd, R. H., Schnabel, R. B., and Schultz, G. A.: "*Parallel Quasi-Newton Methods for Unconstrained Optimization*," Mathematical Programming, 42, 1988, pp. 273-306. 167

[9] Chang, K. J., Haftka, R. T., Giles, G. L., and Kao, P.-J.: "Sensitivity-based scaling for approximating structural response," J. Aircraft, Vol. 30, 1993, pp. 283-288. 157

[10] Coplien, J. O.: *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

[11] Cressie, N.: Statistics of Spatial Data, John Wiley and Sons, New York, NY, 1991. 163

[12] Dennis, J.E., and Lewis, R.M.: "Problem Formulations and Other Optimization Issues in Multidisciplinary Optimization," AIAA Paper 94-2196, AIAA Symposium on Fluid Dynamics, Colorado Springs, CO, June 1994. 119, 165

[13] Dennis, J. E., and Torczon, V. J.: "*Derivative-Free Pattern Search Methods for Multidisciplinary Design Problems*," AIAA Paper 94-4349 in Proceedings of the 5th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept. 7-9, 1994, pp. 922-932. 132

[14] Der Kiureghian, A. and Liu, P. L.: "Structural Reliability Under Incomplete Information," ASCE Journal of Engineering Mechanics, Vol. 112, EM-1, 1986, pp. 85-104.

[15] Eckstein, J., Hart, W. E., and Phillips, C. A.: *"Resource management in a parallel mixed integer programming package*," Proceedings of the 1997 Intel Supercomputer Users Group Conference (http://www.cs.sandia.gov/ISUG97/program.html), Albuquerque, NM, June 11-13, 1997. 150

[16] Eckstein, J., Hart, W. E., and Phillips, C. A.: "PICO: An object-oriented framework for parallel branch and bound," in Inherently Parallel Algorithms in Feasibility and Optimization and their Applications, (eds.) D. Butnariu, Y. Censor, and S. Reich, Elsevier Science Publishers, Amsterdam, Netherlands, 2001. 57, 150

[17] Eldred, M.S., Giunta, A.A., van Bloemen Waanders, B.G., Wojtkiewicz, S.F., Hart, W.E., and Alleva, M.P.: "DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 3.1 Reference Manual." Sandia Technical Report SAND2001-3515W, Updated April 2003. Available online from http://endo.sandia.gov/DAKOTA/software.html 28, 40, 42, 46, 52, 56, 63, 65, 71, 83, 130, 131, 132, 133, 142, 145, 146, 154, 175, 176, 206

[18] Eldred, M.S., Giunta, A.A., van Bloemen Waanders, B.G., Wojtkiewicz, S.F., Hart, W.E., and Alleva, M.P.: "DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 3.1 Developers Manual." Sandia Technical Report SAND2001-3514W, Updated April 2003. Available online from http://endo.sandia.gov/DAKOTA/software.html 61, 161, 191

[19] Eldred, M.S., Giunta, A.A., Wojtkiewicz, S.F., Jr., and Trucano, T.G.: "Formulations for Surrogate-Based Optimization Under Uncertainty," paper AIAA-2002-5585 in Proceedings of the 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Atlanta, GA, Sept. 4-6, 2002. 153, 156, 222

[20] Eldred, M. S., and Schimel, B. D.: *"Extended Parallelism Models for Optimization on Massively Parallel Computers*," paper 16-POM-2 in Proceedings of the 3rd World Congress of Structural and Multidisciplinary Optimization (WCSMO-3), Amherst, NY, May 17-21, 1999. 150

[21] Eldred, M. S., and Hart, W. E.: *"Design and Implementation of Multilevel Parallel Optimization on the Intel TeraFLOPS*," AIAA Paper 98-4707 in Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, Sept. 2-4, 1998, pp. 44-54. 165, 166

[22] Eldred, M. S., Hart, W. E., Schimel, B. D., and van Bloemen Waanders, B. G.: "Multilevel Parallelism for Optimization on MP Computers: Theory and Experiment," AIAA Paper 2000-4818 in Proceedings of the 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA, 2000. 166, 172, 174

[23] Eldred, M. S.: *"Optimization Strategies for Complex Engineering Applications*," Technical Report SAND98-0340, Sandia National Laboratories, Albuquerque, NM, 1998. 13

[24] Eldred, M. S., Hart, W. E., Bohnhoff, W. J., Romero, V. J., Hutchinson, S. A., and Salinger, A. G.: *"Utilizing Object-Oriented Design to Build Advanced Optimization Strategies with Generic Implementation*," AIAA Paper 96-4164 in Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Bellevue, WA, Sept. 4-6, 1996, pp. 1568-1582. 72, 166

[25] Eldred, M. S., Outka, D. E., Bohnhoff, W. J., Witkowski, W. R., Romero, V. J., Ponslet, E. R., and Chen, K. S.: *"Optimization of Complex Mechanics Simulations with Object-Oriented Software Design*," Computer Modeling and Simulation in Engineering, Vol. 1, No. 3, August 1996. 72

[26] Flaggs, D.: "JPrepro User's Manual," in preparation. 186

[27] Friedman, J. H.: *"Multivariate Adaptive Regression Splines*," Annals of Statistics, Vol. 19, No. 1, March 1991, pp. 1-141. 60, 164

[28] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[29] Ghanem, R. and Red-Horse, J.R.: "Propagation of Probabilistic Uncertainty in Complex Physical Systems using a Stochastic Finite Element Technique," Physica D, Vol. 133, 1999, pp. 137-144. 56, 113, 123

[30] Ghanem, R. G. and Spanos, P. D.: Stochastic Finite Elements: A Spectral Approach, Springer-Verlag, New York, NY, 1991. 56, 113, 123

[31] Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H.: *User's Guide for NPSOL (Version 4.0): A Fortran Package for Nonlinear Programming*, System Optimization Laboratory, TR SOL-86-2, Stanford University, Stanford, CA, 1986. 57, 132

[32] Gill, P. E., Murray, W., and Wright, M. H.: *Practical Optimization*, Academic Press, San Diego, CA, 1981. 15, 23, 141, 211

[33] Gilly, D., UNIX in a Nutshell, O'Reilly and Associates, Inc.: Sebastopol, CA, 1992. 93

[34] Giunta, A. A.: "Use of Data Sampling, Surrogate Models, and Numerical Optimization in Engineering Design," AIAA Paper 2002-0538 in Proceedings of the 40th AIAA Aerospace Science Meeting and Exhibit, Reno, NV, January 2002. 157, 159

[35] Giunta, A. A., and Eldred, M. S.: "Implementation of a Trust Region Model Management Strategy in the DAKOTA Optimization Toolkit," AIAA Paper 2000-4935 in Proceedings of the 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA, 2000. 146, 157

[36] Giunta, A. A., and Watson, L. T.: "A Comparison of Approximation Modeling Techniques: Polynomial Versus Interpolating Models," AIAA paper 98-4758 in Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, 1998, pp. 392-404. 60, 163

[37] Goldberg, D. E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wessley Publishing Co. Inc.: Reading, MA, 1989. 42

[38] Gropp, W., and Lusk, E.: *User's Guide for mpich, a Portable Implementation of MPI*, Argonne National Laboratory, Mathematics and Computer Science Division, Report ANL/MCS-TM-ANL-96/6, 1996. 174

[39] Gropp, W., Lusk, E., and Skjellum, A.: *Using MPI, Portable Parallel Programing with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994. 170

[40] Haftka, R. T. and Gurdal, Z.: Elements of Structural Optimization, Kluwer, Boston, MA, 1992. 15, 42

[41] Haldar, A. and Mahadevan, S.: Probability, Reliability and Statistical Methods in Engineering Design, Wiley, New York, NY, 2000. 15, 19, 65, 119

[42] Hart, W.E.: "*Coliny Users Manual. Version 2.0*" Sandia National Laboratories Technical Report SAND2006-xxxx, Albuquerque, NM. Available online from http://software.sandia.gov/Acro/Coliny/ 40, 42, 130

[43] Helton, J. C. and Davis, F. J.: "Sampling-Based Methods for Uncertainty and Sensitivity Analysis," Technical Report SAND99-2240, Sandia National Laboratories, Albuquerque, NM, 2000. 114

[44] Hough, P. D, Kolda, T. G., and Torczon, V. J.: "Asynchronous Parallel Pattern Search for Nonlinear Optimization," Technical Report SAND2000-8213, Sandia National Laboratories, Livermore, CA, 2000. 56, 130

[45] Iman, R. L. and Shortencarier, M. J.: “A Fortran 77 Program and User’s Guide for the Generation of Latin Hypercube Samples for Use with Computer Models,” NUREG/CR-3624, SAND83-2365, Sandia National Laboratories, Albuquerque, NM, 1984. 55, 114, 161

[46] Kernighan, B. W., and Ritchie, D. M.: *The C Programming Language*, Second Edition, Prentice Hall PTR, Englewood Cliffs, NJ, 1988. 72, 74, 79, 169

[47] Koehler, J. R., and Owen, A. B.: “Computer Experiments,” in Volume 13 of the Handbook of Statistics, eds. S. Ghosh and C. R. Rao, Elsevier Science, New York, NY, 1996. 107, 163

[48] Lewis, R. M., and Nash, S. N.: “A Multigrid Approach to the Optimization of Systems Governed by Differential Equations,” AIAA Paper 2000-4890, 2000. 157

[49] McKay, M. D., Beckman, R. J., and Conover, W. J.: “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code,” Technometrics, Vol. 21, No. 2, 1979, pp. 239-245. 114

[50] Meza, J. C.: “*OPT++: An Object-Oriented Class Library for Nonlinear Optimization*,” Technical Report SAND94-8225, Sandia National Laboratories, Livermore, CA, 1994. 36, 57, 132, 142

[51] Meza, J. C., and Plantenga, T. D.: “*Optimal Control of a CVD Reactor for Prescribed Temperature Behavior*,” Technical Report SAND95-8224, Sandia National Laboratories, Livermore, CA, 1995.

[52] Moen, C. D., Spence, P. A., and Meza, J. C.: “*Optimal Heat Transfer Design of Chemical Vapor Deposition Reactors*,” Technical Report SAND95-8223, Sandia National Laboratories, Livermore, CA, 1995.

[53] Moen, C. D., Spence, P. A., Meza, J. C., and Plantenga, T. D.: “Automatic Differentiation for Gradient-Based Optimization of Radiatively Heated Microelectronics Manufacturing Equipment”, paper AIAA-96-4118 in *Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Bellevue, WA, 1996, pp. 1167-1175.

[54] Myers, R. H., and Montgomery, D. C., Response Surface Methodology: Process and Product Optimization Using Designed Experiments, John Wiley & Sons, Inc.: New York, NY, 1995. 110, 119, 162

[55] Nocedal J., Wright S. J.: Numerical Optimization, Springer Series in Operations Research, Springer, New York, NY, 1999. 15

[56] Perttunen, C.D., Jones, D.R., and Stuckman, B.E.: Lipschitzian optimization without the lipschitz constant. Journal of Optimization Theory and Application, 79(1):157-181, 1993. 130

[57] Ponslet, E. R., and Eldred, M. S.: ““Discrete Optimization of Isolator Locations for Vibration Isolation Systems: an Analytical and Experimental Investigation,” AIAA Paper 96-4178 in *Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Bellevue, WA, Sept. 4-6, 1996, pp. 1703-1716. Also appears as Sandia Technical Report SAND96-1169, May 1996. 130

[58] Red-Horse, J. R. and Paez, T. L.: “Uncertainty Evaluation in Dynamic System Response,” Proceedings of the 16th International Modal Analysis Conference, Santa Barbara, CA, February, 1998, pp. 1206-1212. 119

[59] Rosenblatt, M.: “Remarks on a Multivariate Transformation,” Annals of Mathematical Statistics, Vol. 23, No. 3, 1952, pp. 470-472.

[60] Schunk, P. R., Sackinger, P. A., Rao, R. R., Chen, K. S., Cairncross, R. A.: “*GOMA - A Full-Newton Finite Element Program for Free and Moving Boundary Problems with Coupled Fluid/Solid Momentum, Energy, Mass, and Chemical Species Transport: User’s Guide*,” Technical Report SAND95-2937, Sandia National Laboratories, Albuquerque, NM, 1995. 207

[61] Sjaardema, G. D.: *"APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses,"* Technical Report SAND92-2291, Sandia National Laboratories, Albuquerque, NM, 1992. 66

[62] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J.: *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996. 170

[63] Sues, R., Aminpour, M., and Shin, Y.: "Reliability-Based Multidisciplinary Optimization for Aerospace Systems," paper AIAA-2001-1521 in Proceedings of the 42rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Seattle, WA, April 16-19, 2001. 220

[64] Tong, C. H., and Meza, J. C.: *"DDACE: A Distributed Object-Oriented Software with Multiple Samplings for the Design and Analysis of Computer Experiments,"* Technical Report SAND##-XXXX, Sandia National Laboratories, Livermore, CA (draft as yet unpublished). See also http://csmr.ca.sandia.gov/projects/ddace/DDACEdoc/html/index.html. 56, 105, 107, 161, 164

[65] Vanderplaats, G. N.: "CONMIN - A FORTRAN Program for Constrained Function Minimization," NASA TM X-62282, 1973. (see also: Addendum to Technical Memorandum, 1978.) 31, 57, 130

[66] Vanderplaats, G. N.: Numerical Optimization Techniques for Engineering Design: With Applications, McGraw-Hill, New York, NY, 1984. 15

[67] : *DOT Users Manual, Version 4.20*, Vanderplaats Research and Development, Inc. Colorado Springs, CO, 1995. 57, 131

[68] Wall, L., Christiansen, T., and Schwartz, R.L., *Programming Perl*, 2nd ed.: O'Reilly & Associates, Cambridge, 1996. 183

[69] Walton, B.: "BPREPRO preprocessor documentation," online document http://bwalton.com/bprepro.html 186

[70] Wu, Y.-T., Shin, Y., Sues, R., and Cesare, M.: "Safety-Factor Based Approach for Probability-Based Design Optimization," paper AIAA-2001-1522 in Proceedings of the 42rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Seattle, WA, April 16-19, 2001. 220

[71] Swiler, L.P. and Wyss, G.D.: "A User's Guide to Sandia's Latin Hypercube Sampling Software: LHS UNIX Library and Standalone Version" Sandia National Laboratories Technical Report SAND04-2439, Albuquerque, NM, July 2004. 65, 105, 114

[72] Zimmerman, D. C.: *Genetic Algorithms for Navigating Expensive and Complex Design Spaces*, Final Report for Sandia National Laboratories contract AO-7736 CA 02, Sept. 1996. 60, 164

[73] Du,Q., Faber,V. and Gunzburger, M.: " Centroidal Voronoi Tessellations: Applications and Algorithms" SIAM Review, Volume 41, 1999, pages 637-676. 110

[74] Saltelli, A., Tarantola,S., Campolongo,F., and Ratto,M.: " Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models." John Wiley & Sons, 2004. 111

[75] Coello Coello, C. A., Van Veldhuizen, D. A., and Lamont, G. B.: *Evolutionary Algorithms for Solving Multi-Objective Problems*, Kluwer Academic/Plenum Publishers, New York, NY, 2002. 46, 222